Real-Time Multimodal Sensing and Understanding of Complex Physical Environments

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Yurii S. Piadyk

January 2023

Real-Time Multimodal Sensing and Understanding of Complex Physical Environments

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Yurii S. Piadyk

January 2023

Approved:

_____

Department Chair Signature

_____

Date

University ID: N14520737
Net ID: ysp248

Approved by the Guidance Committee:

Major: Computer Science

Cláudio T. Silva
Institute Professor
NYU Tandon School of Engineering

Date

Guido Gerig
Institute Professor
NYU Tandon School of Engineering

Date

Daniele Panozzo
Associate Professor
NYU Courant Institute of Mathematical Sciences

Date

Juan P. Bello
Professor
NYU Tandon School of Engineering

Date

Microfilm or other copies of this dissertation are obtainable from:

UMI Dissertation Publishing

ProQuest CSA

789 E. Eisenhower Parkway

P.O. Box 1346

Ann Arbor, MI 48106-1346

Vita

Yurii Piadyk was born in Lviv, Ukraine, on June 3rd, 1993. He has B.S. in Physics (2014) and an M.S. in High Energy Physics (2016) from Taras Shevchenko National University of Kyiv, Ukraine. While completing his undergraduate and master studies, he attended summer school in 2013 at DESY, Hamburg, Germany, and did three internships at the University of Pierre and Marie Curie in Paris. During his internships, he performed an evaluation of the Associative Memory chip (AMchip) and semiconductor sensors at the test-beam facility of the SPS acceleration ring at LHC in CERN, Switzerland. He started his Ph.D. in September 2016, working in various areas, including computer graphics and optics. His interests have eventually converged on the development of sensor networks and custom tracking solutions. During his Ph.D., he was a visiting researcher at NYU Paris and received NYU Dean's Fellowship in 2016 and NYU Provost's Global Research Initiatives Fellowship in 2019.

Acknowledgements

I am grateful to my mother and family members, who have always believed in me and were always rooting for my success.

I would like to thank my advisor, Cláudio T. Silva, for the support, guidance, and exchange of ideas throughout my Ph.D. studies. Thank you for allowing me to work independently on challenging projects and constantly pushing my abilities to a higher level. I would also like to thank the members of my Ph.D. committee, Daniele Panozzo, Juan P. Bello, and Guido Gerig, for their expertise, ideas, and feedback.

My research would not be possible without some amazing collaborations I have had throughout the years. I would like to thank Yitzchak Lockerman, Carlos Dietrich, Bea Steers, Charlie Mydlarz, Mahin Salman, Magdalena Fuentes, Junaid Khan, Hong Jiang, Kaan Ozbay, Semiha Ergan, Peter Xenopoulos, Jianzhe Lin, Sebastian Koch, Markus Worchel, Marc Alexa, Denis Zorin, and Wenzel Jacob. I would also like to thank the Master students Danna Alamer, Kaiyue Feng, Yipeng Qiu, and Haosheng Jiang as well as interns Chang Ge and Ansh Desai who helped collect the data in rain and fight conditions. And special thanks to Shuya Zhao for project contributions, and Otavio Freitas Gomes for helping me to pull off an impossible timeline for one of the applications.

I'm also grateful to the administrative staff of NYU. Kari Schwartz, Eve Henderson, Lisa Hellerstein, and Susana Garcia have always been eager to support me immediately. And, of course, all this work would be impossible without Ann Messinger's (Borray) i-Buy judo and supply chain mastery.

This work was supported by the following funding agencies: National Science Foundation (NSF awards CNS-1828576, CNS-1544753, CNS-1229185, CCF-1533564,

CNS-1730396, CNS-1626098), C2SMART, a Tier-1 USDOT University Transportation Center at NYU, NYU Global Research Initiatives, and DARPA PTG program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding agencies.

Finally, I thank all my friends who assisted me on this journey. Neel Dey and Jorge One for staying fit, Gromit Chan and Francis Williams for insightful conversations, Joao Rulff for immediate availability, and many others from my newborn family in NYC. And special thanks to Liming Luo for helping me to get through the toughest part.

Yurii Piadyk

January 2023

ABSTRACT

Real-Time Multimodal Sensing and Understanding of Complex Physical Environments

by

Yurii S. Piadyk

Advisor: Prof. Cláudio T. Silva, Ph.D.

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy (Computer Science)

January 2023

Sensor networks have dynamically expanded our ability to monitor and study the world. Their presence and need keep increasing, and new hardware configurations expand the range of physical stimuli that can be accurately recorded. Sensors are also no longer simply recording the data, they process it and transform into something useful before uploading to the cloud. However, building sensor networks is costly and very time-consuming. It is difficult to build upon other people's work and there are only a few open-source solutions for integrating different devices and

sensing modalities. In this work, we introduce REIP - a Reconfigurable Environmental Intelligence Platform for fast sensor network prototyping. REIP's most central tool is an open-source software framework, an SDK, with a flexible modular API for data collection and analysis using multiple sensing modalities. REIP is developed with the aim of being user-friendly, device-agnostic, and easily extensible, allowing for fast prototyping of heterogeneous sensor networks. Furthermore, our software framework is implemented in Python to reduce the entrance barrier for future contributions. We demonstrate the potential and versatility of REIP in real-world applications, along with performance studies and benchmark REIP SDK against similar systems.

We present the following three application examples: (1) An investigation of the correlation between the performance of the heating, ventilation, and air conditioning (HVAC) system in indoor spaces and actual occupancy of these spaces to provide insights into building use patterns for adaptive control strategies of the HVAC system with a goal of reducing building's energy consumption; (2) An urban dataset acquisition using REIP sensors that were built with the support of accurate synchronization. We then demonstrate its utility for pedestrian-vehicle interaction analysis or algorithm development in the context of smart cities where sensor networks empowered by AI techniques provide real-time understanding of the environment for different vehicles, including self-driving cars; (3) The real-time tracking of players in team sports, such as baseball, that provides full skeleton representation for the player over a large game field, which was previously limited to a single point. The system includes independent tracking units with field-of-view adjustable high-speed cameras powered by NVIDIA Jetson and REIP SDK.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sensor networks have expanded our ability to monitor and study the world. They have been used for a wide range of applications, such as monitoring of air pollution [1], urban noise [2] or energy management of smart buildings [3] (Figure 1.1). As their use cases expand, sensor networks become more complex and powerful, enabling a new range of physical stimuli to be accurately recorded, processed, ingested, and analyzed. However, implementing sensor networks is an enormous endeavor with high costs in time and resources. Many decisions have to be made, from which devices to use to which protocols to employ for connecting them. In addition, the deployment and upkeep of sensor networks are critical and time-consuming, which require sophisticated monitoring and alerting tools. Nowadays, it is difficult to build on top of other people's work as there are few accessible open-source solutions suitable for integration into different devices, leading to countless hours of engineering and software design invested every time.

Of note is the case of high throughput sensor applications that incorporate audio or video data capture. Multithreading is typically needed to enable concurrent data

Figure 1.1: Multiple sensing modalities are being actively used to monitor and study the world. From urban noise monitoring with intelligent microphones (left, [2]) to speed limit enforcement with optical or radar sensors (right), sensor networks are actively making our environments more comfortable and safer for living.

capture, processing, and writing to disk. If not handled correctly multiple threads accessing hardware devices or disk locations can lead to race conditions that can result in data corruption or even hardware freezes. Race conditions and hardware lockups can be incredibly difficult to identify and diagnose and are usually only addressed by more experienced developers rather than domain-specific researchers implementing the sensors networks. Sensor networks deployed externally in hard-to-reach locations have a critical need for stability over long periods of time and are particularly sensitive to these kinds of thread-borne failures, which can manifest at arbitrary times, often after many hours, weeks, or even months of operation. The cost of addressing these failures in the field is high, thus, ensuring stable code operation is key.

REIP is a Reconfigurable Environmental Intelligence Platform for fast sensor network prototyping, including an efficient and scalable sensor runtime (Figure 1.2). Given a sensing application and a set of requirements, REIP aims to alleviate the work of designing a remote sensor network by providing tools for sensor node

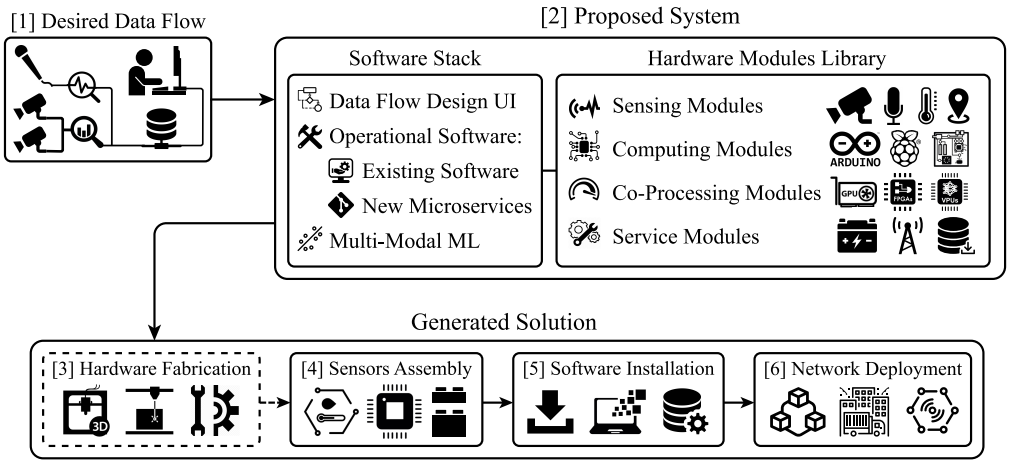


Figure 1.2: REIP platform overview showing stages of operation for users. 1) The researchers determine the sensing needs for their experiment. 2) The platform provides a library of software blocks/components for the implementation of the desired data acquisition and processing pipeline, which are compatible with standard sensing devices/modules (e.g. a USB Audio Class device). 3) At NYU, we have access to prefabricated custom modules, built as part of the design of this instrument. 4) The modules connect to each other with minimum effort, allowing the fast design of full sensors. 5) The software configuration provided by our system can be executed on a variety of computing platforms. 6) The sensors are ready to be placed in the field and activated for data acquisition in a fraction of the typically required design time.

design, software and hardware integration, bandwidth management, and other time-costly aspects. We abstract low-level problems to allow users to focus on their problem-specific challenges.

In this work, we focus on the central tool of REIP referred to as the REIP SDK (or software framework) – a modular API containing a set of re-usable, plug-and-play software blocks that integrate solutions for different hardware and software components by following the best practices. Moreover, we implement this API as a flexible, open-source software framework in Python, with the goal of it being user-friendly and, most importantly, encouraging contributions and extensions in the future. Its block libraries (Section 3.2.1) can also serve as a reference on how to implement different aspects of the sensor software even when users choose/need

to build a custom solution (more details in Chapter 3).

It should be noted that our work is aimed toward building sensors with edge computing capabilities of Linux-based high-performance SBCs (Single Board Computers), such as the NVIDIA Jetson [4, 5] or Raspberry Pi [6]. The amount of data generated by modern sensing platforms (especially those containing video cameras) is such that it is often infeasible to upload all of it to the cloud for later processing. On-the-edge real-time processing of sensor data is required to filter out the background noise or generate more compact representations of the data, and in such scenarios an efficient utilization of the hardware capabilities offered by the computing platform is key. REIP SDK was designed to have minimal performance overhead on such platforms and offers the user full control over the execution of data acquisition and processing pipelines.

The structure of the thesis is as follows: in Chapter 2, we discuss the design of the REIP platform and the corresponding REIP SDK in relation to existing solutions. In Chapter 3, we describe the design approach and API of the REIP SDK, with Section 3.3 focusing on performance studies regarding concurrency and overhead, and report the performance of the REIP SDK on different hardware platforms as well as contrast the performance of our SDK to similar solutions. Furthermore, we present two case studies of smart traffic event detection (Sections 3.4) and object localization and tracking (Section 3.5), and discuss the potential for the platform's use in multi-modal sensing applications (Section 3.6).

To evaluate the practical utility of our contribution we apply REIP in (i) An investigation of the correlation between the performance of the heating, ventilation, and air conditioning (HVAC) system in indoor spaces and actual occupancy of these spaces to provide insights into building use patterns for adaptive control

strategies of the HVAC system with a goal of reducing building's energy consumption (Chapter 4); (ii) An urban dataset acquisition using REIP sensors that were built with the support of accurate synchronization. We then demonstrate its utility for pedestrian-vehicle interaction analysis or algorithm development in the context of smart cities where sensor networks empowered by AI techniques provide real-time understanding of the environment for different vehicles, including self-driving cars (Chapter 5); (iii) The real-time tracking of players in team sports, such as baseball, that provides full skeleton representation for the player over a large game field, which was previously limited to a single point. The system includes independent tracking units with field-of-view adjustable high-speed cameras powered by NVIDIA Jetson and REIP SDK (Chapter 6).

Finally, we discuss limitations and open problems, and a vision of future components of REIP (SDK) in Chapter 7.

1.1 Contributions

The contributions of this work are listed in the following:

1. A design of the REIP platform for fast prototyping of heterogeneous sensor networks;

2. An open-source implementation of the REIP SDK for rapid development of multimodal sensors with edge computing capabilities;

3. Performance evaluation of the REIP SDK under different configurations, including comparison with other existing software frameworks;

4. Extensive benchmarking results from different hardware platforms demonstrating minimal overhead and scalability of the REIP SDK;

5. Two case studies highlighting the utility of the REIP SDK in designing multimodal sensors;

6. A novel solution for synchronization across multiple modalities and sensors;

7. Applications 1: an independent evaluation of NYU's HVAC system performance resulting in the discovery of the presence of energy waste and faulty sensors;

8. Applications 2: a novel dataset with example analysis of urban traffic;

9. Applications 3: a novel modular sports tracking system providing full skeleton representation for each player over the large game field;

10. A motor control technique for faster camera field of view adjustment.

The work resulted in publications of a research article titled "REIP: A Reconfigurable Environmental Intelligence Platform and Software Framework for Fast Sensor Network Prototyping" in the Sensors journal [7], and a utility patent application number US20210287336A1 for sports tracking system (under review). Additionally, a new hardware lab infrastructure was developed in the department of Computer Science and Engineering at NYU Tandon School of Engineering to make these applications possible.

Chapter 2

Related Work

Sensor networks are being used in a large range of applications, each with varying computing requirements and ranging from sensing a single modality with low data loads (e.g., intermittent air quality sensing [8]) to more complex and heterogeneous sensor networks with larger data flows and computing requirements (e.g., audio-visual traffic monitoring [9] or sports analytics [10]). Existing frameworks for sensor network development are typically designed to work on a narrow range of requirements, e.g., low data volumes [11] or large computing resources [12]. As sensor networks become more common and expand their sensing modalities, along with the applications they serve, these frameworks fall short in terms of flexibility and re-usability across different hardware platforms. Table 2.1 provides a feature comparison of various existing sensor network development platforms (top half) as well as software frameworks for building data acquisition and processing pipelines (bottom half). We emphasize that all these projects have their own, sometimes opposite, design goals and can often be complementary to what is proposed in our work. Nonetheless, we try to assess every project on all seven criteria to provide

Table 2.1: Related works feature comparison. The features include (from left to right): open source availability (open); user-friendly API / low barrier for entry; support of multiple sensing modalities; ability to easily add new features or sensing modalities (extensible); ability to handle large amounts of data (scalable); sensor level integration (HW/SW); and whether the framework can be used with various computing platforms and sensor devices (agnostic). The ✓and × symbols denote whether the feature is present or not in each framework. The parenthesis indicates that the feature is present in the given framework but it does not excel at it, or that the framework could possibly be used but it was not designed to have such a feature. A question mark means no assessment.

Project Name	**Open**	**Easy API**	**Multimodal**	**Extensible**	**Scalable**	**HW/SW**	**Agnostic**
FIT-IoT Lab [13]	✓	✓	✓	×	✓	✓	×
FIESTA IoT [14]	×	×	✓	×	✓	✓	✓
Signpost [15]	✓	(✓)	✓	✓	×	✓	×
SensorCentral [16]	×	?	✓	✓	✓	×	✓
AoT [17, 18]	✓	(✓)	✓	✓	×	✓	×
WaspMote [19]	✓	✓	✓	(✓)	×	✓	×
The USC [20]	×	×	?	?	✓	✓	?
FIWARE [21]	✓	(✓)	✓	(✓)	×	×	✓
DDFlow [22]	✓	(✓)	(✓)	(✓)	(✓)	×	✓
EdgeProg [23]	×	?	×	×	(✓)	(✓)	(✓)
Caesar [24]	×	?	×	×	?	×	(✓)
Waggle [25, 26]	✓	✓	✓	✓	×	✓	(✓)
Apache Ray [27]	✓	✓	(✓)	(✓)	(✓)	×	(✓)
Celery [28]	✓	✓	(✓)	✓	×	×	✓
Spotify Luigi [29]	✓	✓	×	×	×	×	(✓)
GStreamer [30]	✓	×	(✓)	(✓)	✓	×	(✓)
DeepStream [31]	✓	×	✓	(✓)	✓	(✓)	×
FFmpeg [32]	✓	×	(✓)	×	(✓)	×	(✓)
REIP (SDK)	✓	✓	✓	✓	✓	✓	✓

as complete an overview as possible. When the framework can not be evaluated directly, because it depends on custom hardware or is not open source, we rely on the results reported in the materials of the corresponding publication. In the following section, we describe the most relevant sensor network platforms, what they excel at, and how they are different from REIP (top half of Table 2.1).

2.1 Sensor Network Platforms

Solutions for the sustainable and reusable development of sensor networks have been explored before in the context of industry as well as in academia. Different alternatives have been proposed, which tackle common challenges such as hardware/software (HW/SW) integration and/or the use of heterogeneous devices, being open-source with user-friendly API, etc.

Among notable platforms with scalable HW/SW integration is FIT-IoT Lab [13], which is a testbed available for researchers addressing wireless communications in sensor networks and low power routing protocols, with embedded and distributed applications. However, it is not device-agnostic, as only a limited set of sensors are supported with no extensibility. Similarly, FIESTA IoT [14] is a meta-testbed IoT/cloud infrastructure designed for the submission of experiments over interconnected hardware testbeds using a single set of credentials. Although it is scalable, it still lacks an extensible and open-source API and thus relies on proprietary testbed deployments associated with the institutions in the FIESTA IoT Consortium.

Other solutions such as Signpost [15], which is an extensible solar energy-harvesting modular platform designed to enable city-scale deployments, are not device-agnostic and only work with the customized sensors they provide. It also requires a considerable amount of engineering to reproduce such a highly customized sensor network. In contrast, SensorCentral [16] is a device-agnostic, multimodal sensor platform but it does not consider HW/SW integration at the sensor level and is not open-source for the research community to use.

The most similar to the REIP platform is the Array of Things (AoT) [17, 18],

an urban sensing system designed to collect real-time data from the environment leveraging a sensor platform called Waggle [25]. The AoT comprises an open-source API with multiple sensing modalities; however, it does not meet the device-agnostic criteria, as it depends heavily on the Waggle platform which itself cannot be easily produced at scale by other institutions/researchers.

Similarly, WaspMote [19] offers a modular hardware and software architecture integration, with an open-source API to create its application pipelines. Some popular use cases for the system include smart cities, water, and agriculture applications, most of which utilize low-bandwidth wireless technologies such as LoRa or ZigBee. WaspMote was designed for low-volume IoT applications on constrained and generally battery-powered edge devices such as Micro Controller Units (MCUs), which are not typically suited for processing high volumes of audio or video data [33]. REIP, in contrast, is targeting higher performance Linux-based SBCs to tackle such data-intensive sensing applications.

The USC testbed [20] currently under construction is to be a scalable HW/SW integrated sensor network with sensors, actuators, and wireless radios to support experimental research on sensing, processing, algorithms, and software for IoT. However, an open-source codebase and user-friendly API has not so far been announced as part of their design goals.

Ultimately, the aim of REIP is to provide a sensor network development platform that meets all of the before-mentioned criteria in a balanced way. As the first step, we implement and present in this work the REIP SDK—an open-source device-agnostic SDK/API, that supports multiple sensing modalities and hardware/software integration, and which is not only user-friendly but also scalable and extensible. The following section details existing software frameworks that are

relevant to the creation of data acquisition/processing *pipelines* such as the ones created using the REIP SDK for sensing platforms with edge computing capabilities (bottom half of Table 2.1).

2.2 Software Pipeline Frameworks

It is natural to develop sensor software as a data acquisition/processing *pipeline* since they typically contain a data source (e.g., a camera), some form of data processing, and, often, a network layer (although some sensors are standalone devices that store data locally). There exists a variety of software frameworks for building pipelines in different application domains but none are out-of-the-box a good fit for building a software stack with real-time performance on sensors with edge computing capabilities, in particular, modern SBCs. Internet of Things (IoT) pipeline frameworks (e.g., FIWARE [21]) are mainly designed to work with multiple streams of small data packets, where the computational cost of data serialization is not a major concern. Other big data frameworks, such as Apache Airflow [34] or Ray [27], are developed for handling large volumes of data across computing clusters, and are not suitable for running in real-time on IoT devices because of large performance overhead and non-trivial compilation steps on embedded systems. Multimedia pipeline frameworks such as GStreamer [30] are designed for highly-efficient multimedia applications, but their implementation and documentation are difficult to understand for anyone who is not an expert, and thus are time-consuming or in some cases not practical to extend to more specialized sensing applications.

We elaborate on different existing software pipeline frameworks below.

2.2.1 IoT Frameworks

IoT software frameworks for building sensor runtimes are mainly designed in a light-weight manner, assuming small packets of data, e.g., sporadic IoT events or low-volume data streams such as temperature measurements [35]. They do not scale well to larger data streams, such as video processing with machine learning on the edge [36].

FIWARE [21] is an open-source solution with a focus on smart city applications. It uses multiple programming languages, and the community provides docker images of various implementations with different run-time requirements. Because of this versatility, extending FIWARE for custom implementation requires full-stack development knowledge of these languages (e.g., Java, Node.js, C++, and Python), which implies a steep learning curve [37]. The limitations of FIWARE have been highlighted by performance evaluations of the platform [38]. Users state a high barrier to entry and the platform shows high performance with low-volume event data but introduces latency when operating over larger-scale wireless sensor networks. DDFlow [22], a visual and declarative programming abstraction, is a significant contribution to heterogeneous IoT networks. Its goal is to provide a flexible programming framework without burdening users with low-level hardware and network details, such as load balancing. The runtime interface utilizes available resources to dynamically scale and map an IoT application similar to EdgeProg [23]. While DDFlow can be used for multi-modal sensing, the library is very high level and does not currently have the capabilities to facilitate high throughput application pipelines [39].

Other solutions, such as Caesar [24], are either not open-source and lack HW/SW integration, or are simply designed to handle small data loads as is the case with

Waggle [26]. In addition, Caesar is limited to activity recognition using cameras as an application only and, while supporting multiple modalities, Waggle relies on the built-in parallelization and data serialization libraries in Python, which makes it impractical to use for high data throughput applications. The REIP SDK, in contrast, offers multiple parallelization and data serialization strategies to best match the application needs (Section 3.3).

2.2.2 Big Data Frameworks

Distributed big data frameworks have gained traction and have been under heavy development in academia and industry for their ability to concurrently process large amounts of data. These frameworks, including Apache Ray [27], Spark [40], Celery [28] and Spotify's Luigi [29], provide many of the concepts we are looking for with regard to the modular design of complex pipelines. However, the intended use case of these frameworks is different in that it focuses on task scheduling, tracking, dependency resolution, and coordination across a cluster of machines. These lead to additional serialization and increased latency that are unnecessary on a single local device, causing difficulty in scaling pipelines to fit constrained devices [41]. Ultimately, they were designed for use on the server side and are much better suited for aggregating the data extracted from sensing platforms, rather than running on them.

The most comparable framework for our target use case is Apache Ray. Ray is a universal API for building distributed applications that enable end users to parallelize machine code across multiple CPUs and multiple machines [42]. The foundational library that Ray is built on is Apache Arrow [43], a data management library focused on the fast movement and processing of large amounts of data, which

includes their own highly efficient array serialization formats [44]. Part of Arrow's offering is a shared memory server called Plasma Store [45], which supports memory mapping on Unix-based devices to minimize the overhead of data serialization in multiprocessing applications. While these tools are very useful for facilitating multi-tasking applications, Ray and other big data libraries were designed to run on larger computing clusters and are too heavy to scale to edge devices. REIP's software framework capitalizes on parts of these existing solutions (e.g., Plasma Store) suitable for efficient sensor development while remaining lightweight and accessible for lower power embedding platforms.

2.2.3 Multimedia Frameworks

Highly popular multimedia libraries designed to handle large streams of video and audio data on a variety of devices are GStreamer [30], NVIDIA DeepStream [31] and FFmpeg [32]. GStreamer is an open-source pipeline/graph-based multimedia framework for complex data workflows, used in a variety of multimedia applications such as video editing, transcoding, or streaming media. GStreamer is multi-platform and has been used reliably in pipelines for decades, but it is written in C and requires low-level programming expertise to extend it by implementing custom components outside of the scope of processing video/audio data types. These characteristics make it fall short to be a viable candidate for an easily-extensible and generalizable framework for the development of full application pipelines for sensor networks [46]. Similarly, FFmpeg which was designed for the processing of video and audio files in a CLI (Command Line Interface) does not offer an API for extending it to other modalities.

Some of the GStreamer limitations have been partially addressed by NVIDIA DeepStream, a scalable framework for building and deploying AI-powered video analytics on the edge. DeepStream provides ready-to-use AI components, such as object detection on video frames, but it follows GStreamer's API; thus, new components cannot be efficiently implemented in a high-level programming language, such as Python, but rather are only used through Python bindings. The framework still presents a steep barrier to entry for beginners and has limited flexibility to extend for applications other than audio-visual. Unlike GStreamer, NVIDIA DeepStream is not cross-platform and is dependent on NVIDIA platforms such as the Jetson family [5].

2.2.4 REIP SDK

Finally, the REIP SDK contributes a unique integration of data-flow programming abstractions and system implementation components that meet the productivity and performance needs of real-time IoT data collection and analysis applications. Many design choices in the REIP SDK were made in response to the challenges faced during the design of sensors and it is intended to accelerate the development of the software and integration of different components in a sensor network. We describe our design in the following chapter.

Chapter 3

REIP SDK

3.1 Approach

REIP seeks to provide a flexible and versatile environment for users to build, extend, reconfigure, and share their application code as they move through the rapid development process of designing sensor networks and other IoT data processing pipelines. In order to facilitate this, the REIP SDK provides a small number of abstractions so that users can take their existing code and integrate it seamlessly into a multi-tasking application (the repository is hosted at `https://github.com/reip-project/reip-pipelines`).

3.1.1 REIP SDK as part of sensor network development

A common workflow for sensor network prototyping using REIP is depicted in Figure 3.1. It contains the following typical steps:

1. Define the project requirements, i.e., sensing modalities, resolution, sampling frequency, etc.

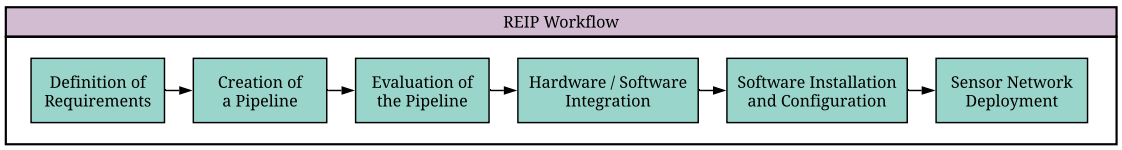


Figure 3.1: Typical steps in the sensor network prototyping process using REIP.

2. Use REIP SDK to build the data collection and processing pipeline. Custom blocks can be defined specific to the project needs, e.g., data processing with machine learning.

3. Evaluate the data collection and processing pipeline and select the optimal edge computing platform.

4. Implement any custom blocks and carry out the sensor build.

5. Install the REIP SDK runtime on all of the edge sensors and the server.

6. Deploy sensor network for data collection and processing.

This chapter focuses on the REIP SDK, its API, performance, and design principles. We discuss other components of the REIP platform that we foresee building (i.e., a simulator to speed up step 3 in the workflow) in Chapter 7.

3.1.2 Benefits of the REIP SDK

When building application code that handles data streaming and processing, it is often beneficial to decouple the program into smaller pieces that can run independently at their own rates using either multithreading and/or multiprocessing, which will prevent computationally demanding parts of the program from inhibiting the rest of the program (e.g., prevent machine learning from blocking video sampling).

However, doing so often requires repeatedly solving and scaffolding the same problems: How should the data be moved around? What serialization method to use? What about error handling? and so on. There is a lot to consider when writing parallelized code, and when one has several applications, re-implementing solutions to the same problem while mixing the parallelization logic with the application logic, it creates software that is very difficult to maintain and reuse in new projects.

The REIP SDK formalizes design patterns that repeatedly emerge in many sensing applications. Typically, one has a collection of workers, each with some initialization, data processing logic, and cleanup, where each of these workers communicates with others via thread-/process-safe queues for data sharing and management. Seemingly simple, such an approach can quickly result in a difficult-to-maintain code base for complex applications, resulting in deadlocks or other issues common to multitasking implementations. It takes a lot of effort and domain knowledge to structure such code properly. The REIP SDK offers a unique approach to making the implementation of data acquisition and processing software fast, easy and reliable in multi-worker contexts.

3.1.3 Design Principles

The REIP SDK seeks to cater to a wide range of domains, programmer expertise, and compute constraints. To that end, we sought to follow these four principles:

Accessibility

One of the important goals of the REIP platform is to provide researchers with a broad range of expertise and backgrounds the capabilities to perform environmental sensing projects. We thus chose the Python programming language for the REIP

SDK because of its wide adoption, shallow learning curve, and a wide ecosystem of libraries spanning countless domains, including: data science and machine learning. REIP's API choices also take design inspiration from popular machine learning frameworks that excel at defining connections between different components, which are already familiar to many engineers working with data.

Extensibility

In order to address the requirements of a diverse set of applications, we designed the REIP SDK in a modular fashion. The atomic component of the framework is a Block, which represents one computational unit (e.g., acquiring an image from a camera or applying an object detection model, etc.) with a variable number of inputs and outputs.

Multimodality

The REIP SDK was designed to make minimal assumptions about the data that is being processed, allowing it to be used in a multitude of contexts. The primary constraint is that the data be serializable for cases where inter-process communication is required. Any domain-specific implementation details are delegated to custom block implementations, which promotes a clean and principled separation of concerns between data engineering and the research problem domain.

Scalability

Python, like many programming languages, provides options for scaling code to run multiple operations at the same time through concurrent programming, commonly known as multi-threading and multi-processing. REIP takes advantage of this by executing each block in its own thread, allowing them to run independently

to minimize the latency and maximize the data throughput. The bottleneck in multi-process Python applications is often data serialization, and we leverage Plasma Store [45] for an efficient shared memory implementation where child blocks can access the read-only version of the data with fixed memory mapping overhead.

3.2 Programming Interface

The REIP SDK takes much of its API inspiration from graph definition in Keras [47] and Scanner [48], and its usage consists of two stages. The first is a computational graph/pipeline definition stage. Here, the user declares all of the blocks that are going to be used in the pipeline, how they are being distributed in a multiprocessing context, and their interconnections, so that they are able to pass data from one to another. An example of this is shown in Figure 3.2b. Note that none of the data processing code is being executed at this stage.

Once the graph is defined, finally, we can execute it. This is done by simply calling *graph.run*(), which will spawn all of the graph's children and begin data processing. The behavior of this stage is all controlled within the block class definition. We elaborate on each of the SDK components in the following sections.

3.2.1 Blocks

A Block is a fundamental component of the REIP SDK, and is implemented as a Python class that represents one unit of computation (e.g., get audio from the microphone, compute machine learning outputs, upload data to server, etc.). Each Block runs in its own independent thread and uses Queues to pass data to others.

Blocks are designed to be easily extendable to suit diverse use cases. A Block

```python
import reip, random
from tensorflow.keras.models import load_model

class ObjectDetection(reip.Block):
    def init(self):
        self.model = load_model('path/to/model.h5') # Initialize the model

    def process(self, *frames, meta):
        i = random.randrange(0, len(frames)) # If a cameras to process
        frame = frames[i] / 255.0 * 2 - 1 # Preprocess image for the model
        predictions = self.model.predict(frame) # Evaluate the keras model
        return predictions, {'frame_index': i} # Return model's predictions

    def finish(self):
        self.model = None # Free the model to be garbage collected
```

(**a**)

```python
import reip.blocks as B

# Graph / data pipeline definition
with reip.Graph() as graph:
    # Read video from two cameras
    with reip.Task('cam0'):
        cam0 = B.video.Camera(device=0)
    with reip.Task('cam1'):
        cam1 = B.video.Camera(device=1)

    # Perform object detection and write to file
    objects = ObjectDetection()(cam0, cam1, throughput='large')
    B.JSONWriter('objects/{time}.json')(objects)

    # Upload files to a server endpoint
    B.upload.UploadFiles('https://myserver.com/api/upload', 'objects/*')

graph.run() # Graph execution
```

(**b**)

Figure 3.2: Code examples illustrating the usage and extensibility of REIP SDK in video capture and processing application. (**a**) Block implementation for a machine learning model. It consists of an *init*, *process*(...) and *finish* methods. The *process* method takes frames from multiple cameras and chooses one to perform object detection on it. (**b**) Graph definition showing object detection on a video stream with the detections saved in JSON files and uploaded to an API endpoint.

consists of: an initialization function $init()$ that is called at the start and which can be used to acquire resources and set initial values; a process function $process(...)$ that is called repeatedly with data from parent blocks as an input and returns 0 or more outputs to the next block(s); and a cleanup function $finish()$ that is called at the end to release any resources acquired. This general program structure encapsulates a wide family of programs and is fundamental in Object Oriented Programming and Python context managers. An example block implementation is shown in Figure 3.2a. Note that for a custom block, any of these functions can be omitted if not used.

Blocks can operate in four approximate roles describing how they relate to the data that they are handling (definitions are not binding):

- Data source (0 inputs, $\geq$1 outputs, e.g., sensing device such as a microphone);
- Data processing ($\geq$1 inputs, $\geq$1 outputs, e.g., object detection in an image);
- Data sink ($\geq$1 inputs, 0 outputs, e.g., data storage to disk);
- Operational (0 inputs, 0 outputs, e.g., disk usage monitoring).

Pipelines can be constructed by connecting a data source block to any number of data processing and/or sink blocks. Operational blocks are typically considered standalone blocks that perform operations without needing to communicate with any other blocks, e.g., a block that monitors and maintains the network connectivity or available disk space.

Users are also able to customize the rules around when a block will execute. For example, a user can customize the strategy used to determine when the framework should call the process function based on the input queue status. So one can change

whether a block needs to wait for all inputs to have a value or if it should be executed when at least one of them has a value. A user can also configure the max rate at which a block will run, or the strategy used to get items from the queue, e.g., should a block process the latest value in the queue only or process every buffer.

In its current state, the REIP SDK offers dozens of blocks covering audio tasks (recording, SPL computation, etc.), video tasks (recording, pose, object or motion detection), data output, data encryption, data upload, and general utilities. They are organized into corresponding block libraries that serve two main purposes. The first is to speed up the development of sensing applications by means of reusing pre-existing blocks for common tasks. The second, the more subtle, benefit of having libraries of blocks that follow a standardized design pattern is documentation of how to perform various tasks in the sensor network building context. We believe that community contributions will greatly extend the range of supported sensing modalities and operations that can be executed on acquired data.

3.2.2 Graphs

Any interesting application will consist of multiple blocks connected together. In order to control multiple blocks at once, they can be assigned to a Graph, allowing them to be spawned, joined, and managed together. This joint management of blocks also allows the framework to coordinate when one block experiences errors, the others can either continue running, pause or shut down.

Adding blocks to a graph is very easy (see Figure 3.2b) and involves simply defining the block inside of the graph's context (i.e., define them indented under the *with* statement).

3.2.3 Tasks

By using Blocks and Graphs, we are able to define a data processing pipeline that utilizes multi-threading in a single process. However, a single Python process is constrained by Python's Global Interpreter Lock (GIL), which allows for the parallelization of the IO-bound code, but not CPU-bound code. In order to utilize all of the available CPUs efficiently, we need to use multi-processing to spawn multiple Python processes (with independent GILs) that can distribute the blocks to run on all of the CPUs. For this, the REIP SDK provides a special type of a Graph, called Task, that works much like a Graph except that all blocks added to it will be executed in a subprocess controlled by that Task object. They have an identical usage as can be observed in Figure 3.2b.

Blocks are able to detect when their connections are spanning different tasks, meaning that the end user does not need to worry about the logistics of passing data between processes. Users have the ability to specify the type of serialization to use that best suits their data type and volume, which provides flexibility and the opportunity for optimization. Communication with Apache Arrow Plasma Store [45] is built into REIP's cross-task data passing and can be enabled by passing *throughput = "large"* when defining a connection between blocks. It provides efficient, high-volume data throughput where required (Figure 3.2b). Other serialization options include the standard Python Pickle method (low throughput) and Apache Arrow's default serializer (medium throughput).

Error handling is another problem for multiprocessing and multithreading and, by default, it is difficult to report errors back to the main process/thread. The REIP SDK handles this within Blocks and Tasks for the user and will raise unhandled Block and Task exceptions in the main thread/process.

3.2.4 Data Formats

If a user has a specific sensing problem, they can easily extend the REIP SDK by following the I/O specifications between blocks. Input and output buffers consist of an arbitrary data payload and a dictionary with metadata:

```
buffer = (data, metadata)
```

`Data` is the primary data payload, and is commonly (though not necessarily) a Numpy array. By convention, if an output array contains a temporal dimension, then it should appear first and channel information should appear last. For example, video clips would have the dimensions $[Time, Height, Width, Channel]$, and audio data would have the dimensions $[Time, Channel]$.

3.2.5 User-Defined Blocks

Figure 3.3 illustrates the process of implementing a data source block in a seismic sensing application (new modality). The user can focus on the interaction with the sensing device (sample readout) when implementing this block and, after the conversion of the data format to comply with the REIP API, can immediately get access to and benefit from the vast REIP SDK infrastructure (Figure 3.3b). A generic Rebuffer block is used to aggregate the samples into batches that can be processed using a re-purposed STFT (Short-Time Fourier Transform) block from the audio library. The rest of the functionality needed to produce a fully functional sensor, such as data storage to disk and data upload, is also available in the REIP SDK. Additionally, with a single line of code, the SeismicSensor block can be put into the context of a Task to ensure that the computationally intensive STFT block does not interfere with any data readout operations that need to be performed at a

```python
import reip, serial, numpy

class SeismicSensor(reip.Block):
    port = "default" # Serial port connection
    rate = 100 # 100 Hz sensor sampling rate

    def __init__(self, **kw):
        super().__init__(n_inputs=0, **kw) # A data source block has no inputs

    def init(self):
        self.dev = serial.Serial(port=self.port) # Initialize the connection

    def process(self, *xs, meta):
        raw = self.dev.read(3*2) # Acquire sensor reading (xyz @ 16 bit)
        # And format the data as per REIP API (axis=0 is for time)
        data = numpy.frombuffer(raw, dtype=np.int16).reshape(1, -1)
        return data, {'sr': self.rate} # Pass the data to the next block

    def finish(self):
        self.dev.close() # Close the connection with sensing device
```

(**a**)

```python
import reip.blocks as B
from seismic import SeismicSensor

with reip.Graph() as graph:
    # Capture data in an independent process using Task to prevent data loss
    with reip.Task('Sensor'):
        sensor = SeismicSensor(port='/dev/ttyS0')

    # Compute signal magnitude for 1 sec chunks and store results in CSV format
    magnitutes = sensor.to(B.Rebuffer(duration=1)).to(B.audio.Stft())
    B.CSVWriter('magnitudes/{time}.csv', max_rows=100)(magnitutes)

    # Upload data to a server endpoint
    B.upload.UploadFiles('https://myserver.com/api/upload', 'magnitudes/*')

graph.run() # Execution the pipeline
```

(**b**)

Figure 3.3: Code example illustrating how to extend REIP SDK to work with new sensing modality. (**a**) Block code for capturing seismic sensor readings, formatting them according to REIP API, and passing the data processing pipeline downwards. (**b**) Graph definition code demonstrating how to capture a new modality while reusing existing REIP blocks (e.g., Short-Time Fourier Transform (STFT)).

high rate. This is achieved by means of execution of the SeismicSensor block in a separate process managed by the task, with data passing between these processes handled transparently by the REIP SDK. All this functionality is achieved with less than 30 lines of code (comments excluded).

3.2.6 Data Security

There can be many concerns around data security when it comes to IoT devices, whether it be about remote access to the devices, interception of data upload, or direct access to data storage cards. Most of these concerns are outside the scope of REIP SDK, as addressing them requires OS-level handling. Many can be circumvented though by thoughtful system design, such as protecting outside connections to the devices using firewalls, SSH keys, VPNs for secure remote access etc.

A harder-to-remedy issue around IoT (or any unaccompanied computer system for that matter) is that a hard drive is non-trivial to secure. For example, IoT devices often need the ability to reboot themselves in the case of system failure or power interruptions. This poses problems when trying to fully encrypt the hard drives because a password login would be required whenever there was a reboot. The other option is physical security, i.e., making the SD card more difficult to access; however, it also makes it more difficult, and potentially costly, to repair the sensors. Therefore, encryption of sensitive data using a two-sided encryption key is important to ensure that the data cannot be directly accessed from the hard drive itself and can only be decrypted by the main server where the decryption key is stored securely. REIP SDK provides blocks for this type of encryption, including a two-stage encryption technique that reduces network bandwidth for decryption on the server by only needing to transmit a small payload instead of the full data.

3.3 Performance Evaluation

In this section, we implement a number of benchmarks to evaluate the performance of the REIP SDK in different application scenarios. First, we implement a video processing pipeline to understand the performance of the concurrency tools offered by the REIP SDK (Blocks, Tasks, etc.) under different configurations (Section 3.3.1). We also take a closer look at the serialization strategies available (Section 3.3.1.2), and in which configurations they are the most beneficial, as well as compare the performance of the REIP SDK to existing frameworks (Section 3.3.1.5). An audio processing pipeline is then evaluated on different hardware platforms (Section 3.3.2) to estimate the performance overhead of the REIP SDK and to identify the optimal hardware platform (Section 3.3.2.2) for building a physical sensor prototype powered by the REIP SDK that will be used in our case studies (Sections 3.4 and 3.5).

3.3.1 Concurrency

One of the typical problems that users face when building sensors is processing data in real time. When using Python to create a data processing pipeline, this will inevitably involve multi-processing and inter-process communication (Section 3.2.3), where serialization to pass objects from one process to another can be incredibly resource intensive, particularly for large data arrays. With the REIP SDK's concurrency tools, such as Tasks, the user can easily implement a pipeline that meets their project's requirements. The focus of the following subsection is on the evaluation of these concurrency tools under different pipeline configurations.

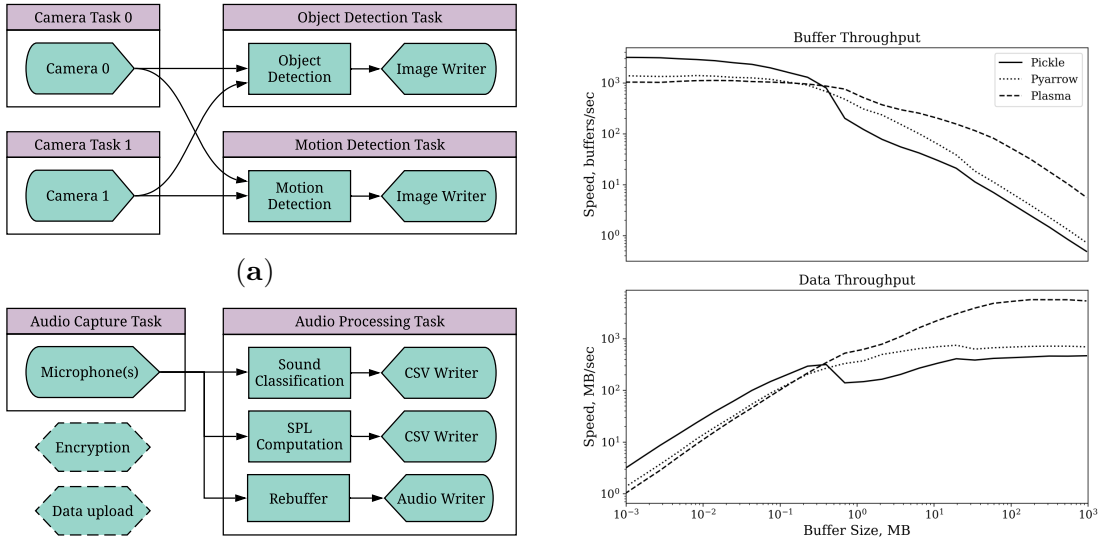


Figure 3.4: Data processing pipelines used to benchmark REIP SDK (left) and comparison of different serialization strategies. (**a**) Video processing pipeline for investigation of REIP SDK performance overhead in different configurations (Table 3.1). (**b**) Audio processing pipeline for comparison of the overall performance of REIP SDK on different platforms (Table 3.2). (**c**) Data throughput between two blocks as a function of buffer size and serialization strategy for inter-process communication on Jetson Xavier NX. Pickle serialization has the lowest overhead for small buffers whilst the Plasma method provides the highest data throughput for larger buffers.

3.3.1.1 Test pipeline

We evaluate the concurrency tools of the REIP SDK using a video processing pipeline (Figure 3.4a) consisting of data acquisition with two camera blocks (5 MP, 14.4 fps) followed by the real-time object and motion detection blocks, and the corresponding image writer block.

Different variations of the video pipeline were executed on the NVIDIA Jetson Xavier NX platform (up to 6 CPUs with 8 GB of RAM and 384 CUDA cores) for 30 s, and the number of frames processed by each block was measured (Table 3.1).

Object and motion detection blocks use the 'latest' connection strategy, while image writer blocks process every frame provided to them by object or motion detection blocks. The motion detection block is outputting one difference image for every two consecutive input frames. The object detection block is next overlaying the bounding boxes of the detected objects on every processed frame and outputs it to the image writer block. We also report the number of lost frames that have not been pulled by the camera block in time for scenarios where the system gets overloaded, as well as any frames still left in the input queues of the image writer blocks. For stereo configurations, object and motion detection blocks are alternating their input on every call of the *process* function.

3.3.1.2 Serialisation

The REIP SDK includes a few serialization methods for the user to choose from to meet their bandwidth requirements. The first method is the most common and uses Pickle, a package in Python's standard library that is capable of serializing arbitrary Python objects. Looking at Figure 3.4c, we can see that Pickle offers the fastest speeds for small buffers (< 0.3 MB, throughput = "small" when connecting blocks), but Apache's Plasma Store [49] ("Plasma" in the plot) is fastest for larger buffers (throughput = "large"). The Plasma Store is a shared memory server that uses Apache Arrow to serialize and provide fast, copy-free memory views of the data that are much more efficient for large data arrays (approaching the memory speed limit).

The final serialization method, Pyarrow, uses the same serialization method as Plasma Store, but it does not provide shared memory (throughput = "medium"). It has a slight performance increase over Pickle for buffers greater than 0.5 MB in size.

See Table 3.1 for a comparison of the impact of different serialization strategies on the overall performance of the video processing pipeline (REIP Hybrid).

3.3.1.3 Configurations

The following pipeline configurations are being evaluated:

- *REIP Hybrid*: image writer blocks are executed in the same task as the corresponding object and motion detection blocks. This is our primary configuration as depicted in Figure 3.4a.
- *REIP Multiprocessing*: each block is executed in an independent task (a full multiprocessing configuration).
- *REIP Threading*: all blocks are being executed in the same (main) process but, by design, in their own independent threads.

We further customize the configurations by using different serialization strategies (Pickle/Pyarrow/Plasma) for inter-process block connections. The system is also intentionally throttled by enabling only four CPU cores to measure smaller variations in performance overhead under these different configurations. For the fairest comparison with other representative frameworks (Ray and Waggle), we implement a thin wrapper emulating the REIP API for each framework (more details in Section 3.3.1.5). We use the same wrapper for the REIP SDK too (REIP Backend in Table 3.1), which does not include different connection strategies or other advanced features of the REIP SDK (hence a slightly better performance compared to the full REIP SDK).

Table 3.1: Performance of different framework configurations when running a video processing pipeline (Figure 3.4a) on Jetson Xavier NX. Values denote the number of frames processed by different blocks in the pipeline during the 30 s sampling period. The top half of the table corresponds to the mono and the bottom half to the stereo camera configuration. The system was throttled to use 4 CPU cores only to be able to measure more subtle performance differences between different configurations. Negative queued values indicate extra frames processed that have already been in the queue prior to the sampling interval. Symbol ˆ indicates queue overflow (max queue size was set to 100 buffers).

Configuration		**Camera 0**		**Camera 1**		**Object Detection**			**Motion Detection**		
Name	**Serialisation**	**Pulled**	**Lost**	**Pulled**	**Lost**	**Detected**	**Saved**	**Queued**	**Detected**	**Saved**	**Queued**
REIP Hybrid	Pickle	433	0	-	-	206	206	0	191	95	0
REIP Hybrid	Pyarrow	431	0	-	-	229	229	0	196	98	0
REIP Hybrid	Plasma	431	0	-	-	351	204	54ˆ	160	80	0
REIP Multiprocessing	Pickle	314	118	-	-	101	73	28	104	52	0
REIP Multiprocessing	Plasma	402	30	-	-	385	184	45ˆ	142	71	0
REIP Multithreading	-	432	0	-	-	388	293	56ˆ	193	96	1
REIP Backend (Mono)	Plasma	431	0	-	-	420	318	34ˆ	422	212	0
Waggle Backend (Mono)	Pickle	410	22	-	-	224	132	52ˆ	202	113	−12
Ray Backend (Mono)	Plasma	432	0	-	-	-	-	-	146	73	0
REIP Hybrid	Pickle	343	89	346	86	126	133	−7	115	57	0
REIP Hybrid	Pyarrow	409	23	321	111	123	122	1	134	66	0
REIP Hybrid	Plasma	327	105	375	57	294	135	39ˆ	181	90	1
REIP Multiprocessing	Pickle	268	164	325	107	79	66	13	82	41	1
REIP Multiprocessing	Plasma	306	126	299	133	255	121	60ˆ	208	104	0
REIP Multithreading	-	432	0	432	0	401	263	50ˆ	346	173	0
REIP Backend (Stereo)	Plasma	433	0	433	0	376	230	64ˆ	318	159	0
Waggle Backend (Stereo)	Pickle	374	58	356	76	102	86	16	64	32	0
Ray Backend (Stereo)	Plasma	432	0	432	0	-	-	-	149	75	−2

3.3.1.4 Performance

The performance metrics in this experiment are the number of frames/buffers processed by each block in the pipeline, so the higher the number in *Pulled* (number of frames successfully acquired by a *Camera* block), *Detected* (number of frames processed by an *Object* or *Motion* detection block), and *Saved* (number of images written by the respective *ImageWriter* blocks) columns in Table 3.1, the better the performance of the pipeline. Conversely, the target values for the *Lost* (number of frames missed by *Camera* blocks due to system overload) and *Queued* (number of frames processed by *Object/Motion* detection blocks but still pending to be saved by the *ImageWriter*) columns are 0 for optimal performance. It should also be noted that the values in different columns are not directly comparable in isolation. The pipeline configuration that processed less of the captured frames but saves more of the processed results can be argued to perform better than the one processing everything but saving little of the results. We provide a complex analysis of the measured metrics in the remaining sections.

It is apparent from Table 3.1 that all stereo multiprocessing configurations are not capable of processing all camera frames in the throttled scenario, with a clear trend of higher performance for higher throughput serialization. Nevertheless, the hybrid approach does succeed in this task for single-camera video. The threading-only configuration shows the highest performance as it does not need to incur any additional performance overhead due to serialization, with the maximum CPU resources spent processing and saving the images (this does not mean that such a configuration is optimal in a general case). Image writing is largely limited by the maximum disk write speed and the remaining CPU resources available after object/motion detection. Some configurations (e.g., REIP Multiprocessing) are less

efficient in data management and result in less of the results being saved (compared to REIP Hybrid) despite high processing rates.

It should be noted that it is only in this specific data acquisition and processing pipeline (Figure 3.4a), where both cameras are operating at the same frame rate (14.4 fps) and object/motion detection blocks are highly optimized (object detection is performed on GPU), that we do not observe data loss for camera blocks due to the Python GIL in the threading only configuration. Data loss would be inevitable when using data sources with significantly different sampling rates. For instance, the microphone block requires being serviced more often than the camera block and will experience overrun errors with data loss due to the Python GIL if there is another block (in the same task/process) performing a heavy computation that does not release the GIL in a timely manner. This makes the Task feature of the REIP SDK essential for the easy decoupling of data source blocks from data processing blocks by placing them into a dedicated task/process with an independent Python interpreter.

3.3.1.5 Comparison with Other Frameworks

To compare the REIP SDK with other existing frameworks, such as Ray or Waggle, we implemented a thin wrapper layer to interface the blocks' implementation with different execution backends (Table 3.1). For REIP, we are using a high throughput hybrid configuration of the video processing pipeline (Figure 3.4a). For Waggle, we map each task in the pipeline to a Waggle plugin and since the Waggle framework does not provide concurrency tools, we use a standard multiprocessing queue to transfer frames between different plugins. In the Ray backend, we use *ray.remote* futures for parallel processing.

Table 3.1 shows that the performance of the REIP backend is higher than the large throughput hybrid configuration executed with the REIP runtime, closer to REIP Multiprocessing. This is expected because the wrapper layer does not provide all of the features of the REIP SDK (e.g., detailed statistics and error handling), which introduces extra overhead.

In turn, the performance of the Waggle backend is similar to the REIP Hybrid configuration with small throughput because the same serialization method (Pickle) is being used. It is a bit lower though (stereo configuration in particular) because of a lack of multithreading in the object and motion detection plugins/tasks.

Finally, Ray provides great concurrency tools but they were developed with different design constraints in mind, which is reflected in its performance. Because Ray uses futures and lazy/deferred evaluation, we observe an execution pattern, where a number of jobs are being accumulated and then executed as a batch that effectively halves its performance. We were also not able to get a GPU-accelerated object detection block working with the Ray backend because the Ray framework does not recognize the Jetson's GPU.

3.3.2 Overhead

Another important aspect of any software framework is its performance overhead. Scalability is one of the design principles of the REIP SDK and for that, its internal routines have been optimized to minimize the amount of service time and maximize the computational resources available for the execution of user code. The percentage of time spent by the framework performing data management and other service routines also depends on the particular computing platform in use, which can vary in the amount of RAM available, a number of CPU cores, and their speed. The

focus of this subsection will be on the performance overhead of the REIP SDK on different hardware platforms.

3.3.2.1 Test Pipeline

We have implemented an audio processing pipeline (Figure 3.4b) that resembles a real-world noise pollution monitoring sensor network (The Sounds Of New York City project or SONYC [50]), excluding the data encryption and upload functionality (bottom-left blocks with dashed lines in Figure 3.4b). In this setup, a mono microphone is recording one-second long audio snippets that are supplied to the sound classification and sound pressure level (SPL) computation blocks for real-time audio analysis. The results are saved to CSV files alongside the raw audio data in 10 s intervals after rebuffering. The most computationally demanding is the sound classification block, where we use a CPU-only implementation of a neural network-based classification model to make the test pipeline compatible with a wider range of computing platforms that may not have GPU support. We have also placed the microphone block in its dedicated audio capture Task to avoid any overrun errors, as explained in Section 3.3.1.4.

3.3.2.2 Hardware platforms

The REIP SDK is Unix-platform compatible so it can be installed and executed on a large variety of single-board computers (SBCs) at various price and power points. This flexibility allows the software to be matched with suitable hardware based on the computational demands of the blocks used in the pipeline and the project budget. We explore the performance of a real-time, high-resolution audio processing pipeline (Figure 3.4b) on different multi-core CPU SBCs (Table 3.2).

Table 3.2: Time spent by different blocks in the audio processing pipeline (Figure 3.4b) performing data processing, idle waiting, and servicing the data between blocks. Comparison is given for different embedded platforms ranging from a low-budget Raspberry Pi 4B to the high-performance NVIDIA Jetson AGX Xavier. The service times remain well under 10 % for each platform, which indicates negligible performance overhead of the REIP SDK. No dropped buffers were detected.

Block	**Raspberry Pi 4B**			**Jetson Nano**		
	Process	**Wait**	**Service**	**Process**	**Wait**	**Service**
Microphone	0.39%	92.9%	6.68%	0.19%	92.4%	7.28%
Machine Learning	42.1%	50.4%	5.12%	30.0%	61.7%	7.73%
SPL Computation	4.74%	88.5%	6.75%	2.58%	90.0%	7.36%
Audio Writer	1.82%	91.6%	6.58%	0.41%	92.4%	7.20%

Block	**Jetson TX2**			**Jetson AGX Xavier**		
	Process	**Wait**	**Service**	**Process**	**Wait**	**Service**
Microphone	0.10%	93.1%	6.72%	0.19%	92.2%	7.33%
Machine Learning	37.7%	56.4%	5.31%	12.2%	78.3%	8.75%
SPL Computation	2.01%	91.1%	6.84%	2.40%	90.0%	7.53%
Audio Writer	0.09%	93.3%	6.63%	0.14%	92.6%	7.25%

3.3.2.3 Performance

The key performance metrics are the percentage of time spent by each block performing data processing, awaiting the next buffer and the service time of the framework managing the data delivery between different blocks. We also record the number of dropped buffers in the sink queues of each block output to identify performance bottlenecks in the pipeline (if any). The criteria for selecting the computing platform is its ability to execute the pipeline at greater than real-time.

Results in Table 3.2 show that even a low-end embedded platform, such as the Raspberry Pi 4B, is capable of processing all the data in real-time using the REIP SDK without dropping a single buffer. The service time overhead of REIP SDK remains well under 10 % on all platforms, which is negligible.

3.3.3 Summary

Real-time data capture and edge processing present many challenges in terms of managing the compute performance and overall data transfer on the device. Existing software frameworks in this domain are often designed with assumptions that data transfer overhead, as a result of serialization, is negligible compared to the anticipated computational load or are designed to be used on large compute clusters without the computational or concurrency constraints inherent in lower power edge sensors. In this section, we have demonstrated that the REIP SDK can handle complex data processing pipelines that outperform other approaches, such as Python's multiprocessing or other frameworks (e.g., Apache Ray).

An important and often overlooked aspect of data processing pipelines is the moving of data between processing blocks. Serialization of data so that it can be exchanged between these blocks can introduce significant overhead in complex pipelines, especially when these data are large in size. REIP's hybrid approach to serialization shows comparable or better performance than other frameworks in the complex video processing pipelines presented. The ability to define the "small", "medium", and "large" throughput flags (defined in Section 3.3.1.2) when establishing block connections provides a way to select the optimal serialization technique applied for the scale of the data being passed between these blocks.

When implementing an audio-based processing pipeline on various embedded computing platforms typically used in sensor networks, service times stayed below 10% when using the REIP SDK. The consistency of these service times is worth noting, as they stay relatively steady between hardware platforms, which suggests that the serialization approach of the REIP SDK is agnostic of the platform it is running on. These service times are likely driven mainly by the memory architecture

of the device which impacts serialization, rather than its computational power. We consider the service overhead of the REIP SDK in these examples to have a minimal impact on overall performance and generalize well to different hardware platforms; however, further work is needed to compare the service times of other frameworks.

In REIP, we are targeting the design, and more specifically in this work, the code implementation and computational challenges that often arise during the design, development, and deployment of individual sensors as well as sensor networks. The REIP SDK aims to provide the user with the tools necessary for the efficient utilization of limited computing resources while remaining easy to use and flexible in development. Let's look closer at how REIP SDK fulfills these tasks in a couple of case studies, with a more in-depth evaluation in real-world application examples (Chapters 4, 5, and 6).

3.4 Case Study: Smart Traffic Event Detection

To showcase the initial implementation of the REIP SDK, we present a case study using a custom multimodal sensor design for real-time traffic analytics with a focus on bicycle accidents. This sensor network is designed for long-term deployment on light poles at busy urban intersections, where power is available but high-speed Ethernet or Wi-Fi-based internet connectivity is not. Researchers want to understand more about the frequency of bicycles that pass through the intersection but also want to analyze video and multi-channel audio of any close interactions between bicycles and other vehicles on the road to study the circumstances around bicycle accidents and the use of multimodal sensor systems to detect these events.

This section focuses on the development workflow steps when using REIP as described in Section 3.1.1. In this case study, steps 3 and 5 have already been addressed through the evaluation of the processing pipelines with hardware selection and the installation of all the necessary runtime software. Due to budget constraints, we have chosen the cheapest of NVIDIA's SBC range with GPU support which, due to the computational requirements of the presented case study, limits us to using only one of the two cameras available in the sensor prototype. Since it is also sufficient for this case study to use one sensor prototype instance, step 5 in the REIP workflow (Figure 3.1) can be skipped, but that by no means limits the number of sensors that can be used in a sensor network powered by the REIP SDK. The more thorough evaluation of sensor hardware and the experience of researchers is a subject of future studies.

3.4.1 Definition of Requirements

For this use case, the following functional requirements must be satisfied by the application pipeline:

1. Video capture with quality higher than 720p at 15 fps;
2. Multichannel audio capture;
3. Object tracking of vehicles;
4. Transmission of traffic analytics and raw multichannel audio and video of the accident near misses via limited cellular plan.

A major constraining factor in this application is the reliance on data-limited cellular connectivity. This excludes the possibility of retrieving continuous video

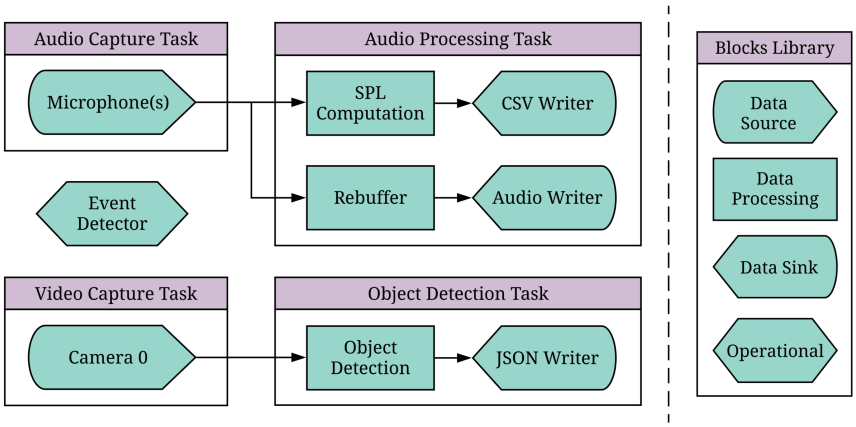


Figure 3.5: Multimodal smart traffic event detection pipeline. This pipeline handles both audio and video. It captures video, passes it to an object detection block, and then writes the detection outputs to JSON files. It also captures audio at short intervals and computes SPL at the one-second resolution, as well as accumulates the audio into longer clips (10 s) and writes them to disk as audio files. Finally, the salient event detection block monitors these files, preserving only those that had an event of interest detected in them.

and multichannel audio data for the post hoc detection of accident events, which would exhaust cellular plan data limits of ≈50 GB/month very quickly. To alleviate the impacts of this constraint, we can leverage recent advancements in a compact but high-power compute devices to push the event detection processes to the sensor itself, so only salient events are transmitted over the constrained network. This is a common need in longitudinal sensing research, where in practice the large majority of collected raw data contains few or no events of interest.

3.4.2 Implementation of Application Pipeline

To highlight the flexibility of the REIP SDK, we have created an application pipeline that constitutes a multimodal *smart traffic event detector* for urban bicycle accident data collection. Figure 3.5 shows the block diagram of this pipeline, where raw video and multichannel audio uploads are preserved only when a salient event

is detected, which in this case is the near miss of a cyclist with a motorized vehicle. To ensure that the maximum amount of useful data is retrieved, both video and audio cues are used to signify a salient event. For video, object detection block using the MobileNet V2 [51] model is fed with video frames at 4 fps from the GStreamer [30]-based camera block, which is also recording the full speed (≈15 fps) video feed into files. An event detector block independently monitors the output of the object detections and deletes the corresponding video fragments if a bicycle and motorized vehicle were not captured in the same frame for any frame in the video fragment. In cases where the camera with an object detector block struggles to capture an event, such as: under low-light conditions, when an event is off-camera, or when it is occluded, the SPL (Sound Pressure Level) computation block will report a spike in decibel level, caused, for example by a horn blast, impact, or tire screech from emergency braking. This information is also used by the event detection block to decide which data fragments contain events of interest and should be preserved.

3.4.3 Hardware and Software Integration

Using the block design, each data source block has a corresponding physical sensing device (e.g., a camera or a microphone) that connects to the computing platform. Here, we define our customized sensing devices (that are compatible with REIP's default data source blocks) and implement any custom blocks required. The prototypical sensor system shown in Figure 3.6 uses two 5 MP USB cameras providing a 160° horizontal field of view (85° max per camera) and 15 fps recording, satisfying our video capture use case requirement. The computing platform is the NVIDIA Jetson Nano Developer Kit [5], which offers edge intelligence capabilities from its 128-core GPU, quad-core 1.43 GHz CPU, and 4 GB of LPDDR4 RAM. The majority

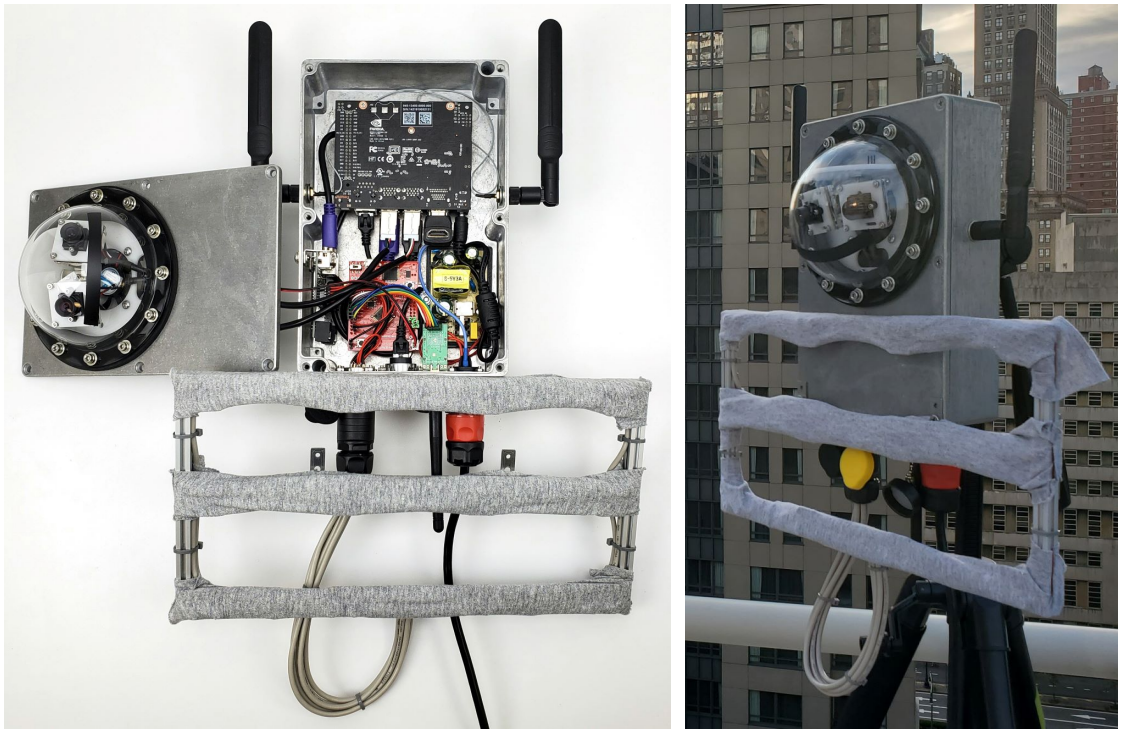


Figure 3.6: Multimodal sensor built for the real-world case study. It includes two 5 MP cameras, a 12-channel microphone array (4 × 3), and a cellular modem. The sensor is waterproof for outdoor operation and powered by the NVIDIA Jetson Nano, which is thermally coupled to the aluminum enclosure for passive cooling.

of the sensor's hardware is enclosed within an aluminum weatherproof housing.

The custom acoustic front-end has been designed to capture synchronized 12-channel audio from its 4 × 3 array of digital pulse density modulated (PDM) Micro Electro Mechanical Systems (MEMS) microphones. It uses the USB MCHStreamer [52] as an audio interface, so the same Microphone block from the REIP audio library can be reused to read data from this USB audio class compliant device. More on the synchronization capabilities of the sensors in Section 3.5.3. In the given application pipeline, a single microphone channel is used as input to the audio processing blocks. For data transmission, a USB LTE modem is integrated containing a SIM card providing a bandwidth-limited cellular plan.

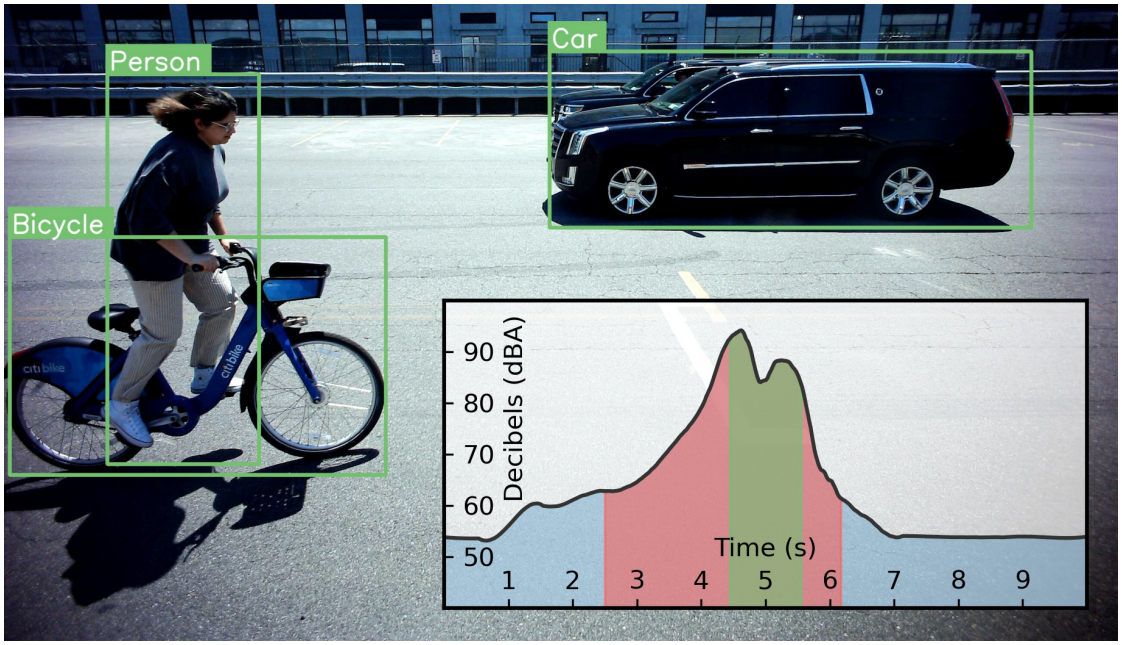


Figure 3.7: Car and bicycle pass-by frame from sensor's camera overlaid with bounding boxes and labels from object detection block for illustration. The green region on the inset audio amplitude plot shows the time period where the car is detectable within video frames, with the red region showing longer periods of audio-based detection of a car horn. The blue portions signify periods when no audio event was detected.

3.4.4 Experimental Deployment

A single sensor was mounted on a tripod at 1.8 meters above the ground facing a roadway in a quiet urban parking lot. This location was chosen to allow for sustained data capture in more controlled conditions than a busy urban intersection. The application pipeline was executed prior to multiple runs of vehicle passes, including (1) motorized vehicle (car, truck, or motorcycle) passes with and without horn use, (2) bicycle and pedestrian passes, and (3) various combinations of simultaneous passes. A total of ≈1 hour of continuous data collection was performed, which would equate to ≈53 GB/h of dual 5 MP video with MJPEG compression and 12-channel audio if all data were to be captured and uploaded. These amounts

of data would overwhelm the majority of cellular data plans, requiring at least 15 MB/s of bandwidth to transmit these data from edge sensor to server, regardless of the inevitable saturation of monthly data plan limits.

Figure 3.7 shows one frame captured from the left video camera for illustration purposes. In this instance, a cyclist was passing by the sensor with two cars driving in the opposite direction while honking their horns. The object detection block has returned confident detections including their bounding boxes around the car, person, and bicycle. The event detection block raised an event flag, as there was the combination of a cyclist and motorized vehicle within the frame, which resulted in the successful upload of 10 seconds of video and multichannel audio of the event via the cellular modem. In the figure bottom, time vs. SPL is plotted with the green region showing the portion of time that the object detector block is able to confidently identify the motorized vehicle passing by, mainly due to the limited field of view of the camera setup. Of note is the much wider red region showing the extent to which the SPL computation block is able to detect the elevated sound level of the car horn, highlighting the advantages of this multimodal approach.

3.5 Second Case Study: Object Localization and Tracking

Urban environments are very dynamic places, but due to deployment logistics, urban research studies often have to grapple with trying to draw inferences from data with low spatial sampling. When dealing with a single sensor covering a large region, it becomes very useful for the sensor to be able to provide spatially dense data of the surrounding area, instead of a single scalar value representing

that whole area. This is the case with a SONYC [50] sensor as it only has one microphone and can only capture a single sound level measurement at a time for the deployment area, which restricts to more macro analyses only.

For vehicular traffic, video-based object detection works well during the day but has a lot more trouble in low-visibility nighttime situations. Sound-based localization provides an alternative way of determining the locations of objects in space without relying on visibility. Using REIP, we built a new *multimodal dense* Wireless Sensor Network (WSN) with the ability to accurately localize sources across multiple sensors using both sounds captured by a microphone array and video captured by two cameras (Figure 3.6).

For this case study, we build the application pipeline, implement the hardware and software integration, and finally, we deploy the WSN at an outdoor testbed for data collection. To evaluate REIP's ability for gathering audio/video data that can be used for localization, we perform two experiments (Figure 3.10): (1) localization of a sound source using multiple sensors in the proximity of the source, and (2) enhanced accuracy localization of an object using the multimodal capabilities of a single sensor.

3.5.1 Definition of Requirements

In this case study, the following functional requirements must be satisfied by the application pipeline:

1. Video recording with quality higher than 720p at 15 fps;

2. Video recording with wide ($> 120°$) field of view;

3. Multi-channel audio capture - up to 12 channels for this application but can

be reduced or increased as needed;

4. Synchronization across data modalities as well as across different sensors;

5. Sound source localization in urban environments;

6. Object tracking of vehicles and pedestrians using multiple data modalities;

7. Disk storage of audio, video, and synchronization data.

With the minimum functional requirements defined, the application pipeline can be implemented using the REIP SDK. This process is detailed in the following section.

3.5.2 Implementation of Application Pipeline

To build the full multimodal pipeline, we create an audio and video processing pipeline depicted in Figure 3.8.

For audio handling, a Microphone block reads audio buffers and outputs arrays in chunks of a fixed size (1 s). To simplify the data flow between peripheral sensing devices, the USB interface is used. The Microphone block uses the ALSA framework which provides audio I/O to the OS from any USB audio class compliant device. We use a USB MEMS microphone as the input device in this example.

A Sound Pressure Level (SPL) computation block takes those short audio chunks, calculates the SPL, and outputs those values to a CSV Writer block which writes to a new CSV file after a fixed number of lines. For on-board machine learning, the audio chunks are fed to a Sound Classification block based on a Tensorflow Lite [53] model which outputs 512-dimension audio embeddings and class prediction probabilities that are piped to a CSV Writer as with the SPL block. The audio

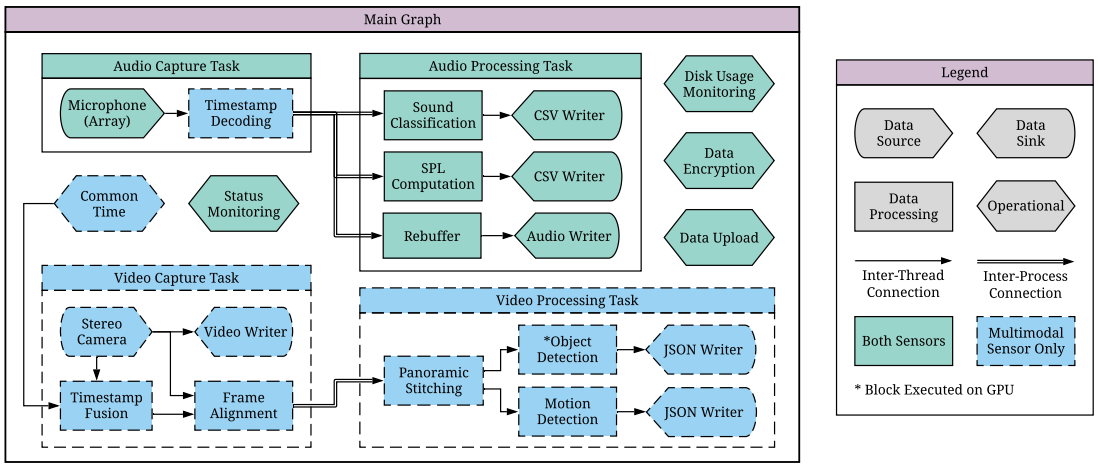


Figure 3.8: Multimodal object localization and tracking pipeline implemented using REIP SDK. Green blocks correspond to the part of the pipeline necessary to (re)implement the functionality of the original SONYC sensor [50]. Blue blocks handle the second video modality added in REIP sensors. Each block is running in its own thread to minimize data processing latency and some blocks (e.g. Object Detection) use hardware acceleration.

chunks are also accumulated into longer segments (10 s) in a Rebuffer block and outputted to a FLAC file. This file is then picked up by the Data Encryption block to ensure audio privacy.

Additionally, the Timestamp Decoding block is analyzing the extra audio channels that contain an embedded synchronization signal (see Figure 3.9) used to align the actual audio data channels with a global common clock as well as timestamp the buffers before processing. This step is essential to do during audio recording to correct for drift caused by small differences in audio sampling frequencies at the audio hardware level. For instance, a typical difference of reference clock frequencies of 10 ppm would result in a drift of approximately one audio sample every two seconds.

To read the video frames from the two cameras, we add a Stereo Camera block that is using GStreamer multimedia framework [54] to simultaneously pull frames from both cameras and have them timestamped using a shared GStreamer clock.

At the same time, the Common Time block is continuously reading a common time timestamp (common for all sensors) from the sensor's micro-controller, which receives this common timestamp distributed by a master radio device shown in Figure 3.10. The two timestamps (common time and GStreamer clock) are then combined by the Timestamp Fusion block and used in the Frame Alignment block to synchronize the video frames with a common, globally synchronized timestamp before processing. Simultaneously, the video streams from the cameras are recorded to file at regular intervals.

There is an optional Panoramic Stitching block that merges (synchronized) images from the two cameras before they are processed by the Object Detection and Motion Detection blocks which, in turn, feed their output to a JSON Writer. The video and JSON files are then uploaded to the server by the Data Upload block similar to the audio upload in the first case study.

We have an Operational-type block that watches the local data directories and uploads files to one of the ingestion servers which are responsible for handling long-term storage. In addition to data upload, we have a Status Monitoring block that will periodically compute statistics about the sensor (CPU load, memory usage, network strength, etc.) and a Disk Usage Monitoring block that will strategically delete files when the disk becomes too full (as would occur during a long period without connectivity).

As described in Section 3.3, our computing platform of choice for this case study is the NVIDIA Jetson Nano, so we can continue with the hardware/software integration where, in addition to defining hardware, we also need to implement a number of custom blocks responsible for the synchronization.

3.5.3 Hardware and Software Integration

Using the block design, each data source block has a corresponding physical sensing device (e.g. a camera or a microphone) that connects to the computing platform. Here, we define our customized sensing devices (that are compatible with REIP's default data source blocks) and implement any custom blocks introduced in the previous section.

3.5.3.1 Sensing Devices

Similar to the previous case study, we use two 5 MP USB cameras fitted to each sensor (see Figure 3.6) allowing up to 160° horizontal field of view and recording at 15 fps, thus, satisfying our use case requirements. A custom acoustic front-end with a 12-channel 4 x 3 microphone array (93 mm horizontal and 72 mm vertical spacing between the microphones) is also satisfying our requirements.

3.5.3.2 Global Synchronization

Adding a common time to the framework requires the implementation of a custom block that will continuously read a timestamp from a USB-connected microcontroller. The accuracy of synchronization is then limited by the resolution of the timestamp provided by the microcontroller and the frequency at which the block is able to update this value (1.2 kHz in our implementation).

The $\approx 0.83\ ms$ accuracy of the common time timestamp provided by the microcontroller via USB is not sufficient to accurately synchronize audio data from different sensors in the same way as video, because audio data is being sampled at a much higher frequency than video (48 kHz vs 15 fps). We propose a novel method of high-precision audio synchronization by means of embedding the synchronization

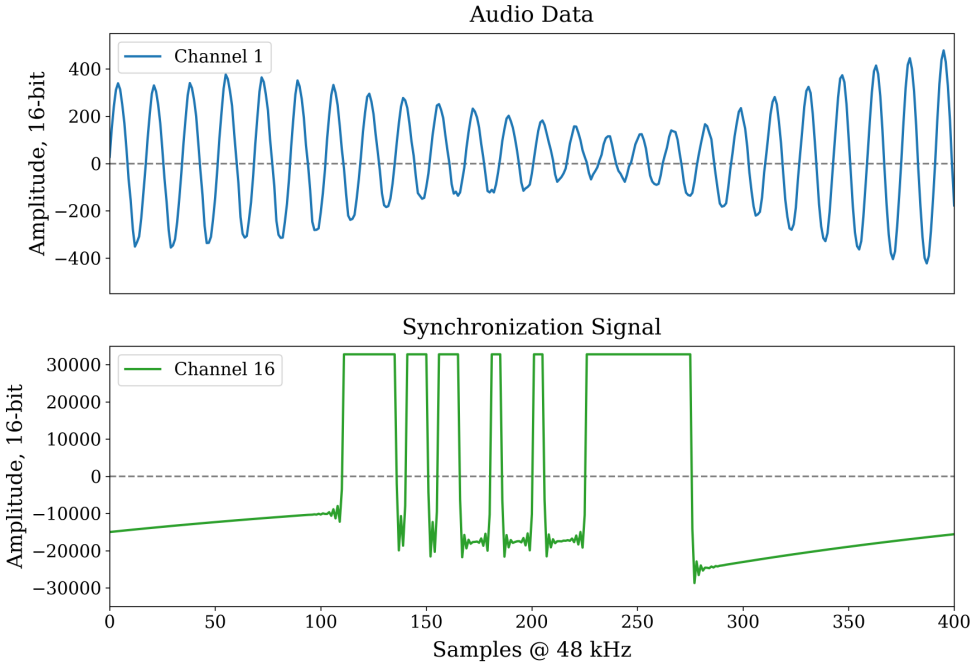


Figure 3.9: Example of synchronization signal embedded into the last channel of audio data at a 120 Hz rate. It contains a serialized 32-bit timestamp that is shared across multiple sensors with 1 μs accuracy using a 2.4 GHz radio module. High synchronization accuracy is required due to high audio sampling rates of 48 kHz.

signal into one of the spare audio channels on the MCHStreamer device, which supports up to 16 channels of Pulse Density Modulated (PDM) audio. An example of how this synchronization appears as PCM audio samples is shown in Figure 3.9. It uses a simple UART-like serial protocol with 1 start bit, a 32-bit payload, and more than 200 audio samples long stop bit. The start bit and payload bits are 5 audio samples wide for more reliable encoding. One audio sample synchronization accuracy is possible because the start bit of the sequence is aligned with the time of arrival of the timestamp from the master radio and the micro-controller has a deterministic delay when processing this information.

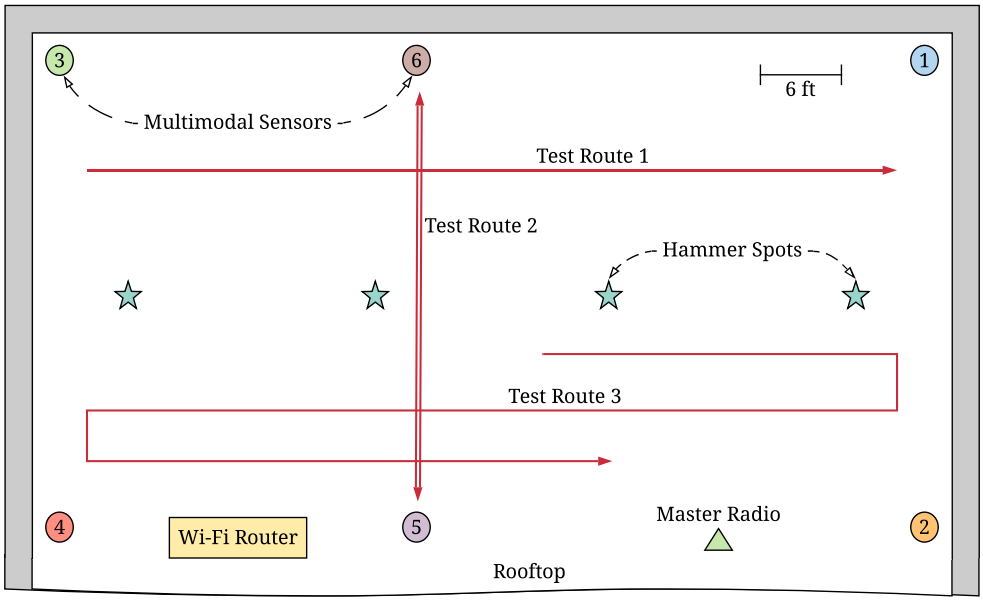


Figure 3.10: The layout (to scale) of a rooftop experiment conducted to evaluate the performance of WSN in the second case study. There were three test routes used for localization using global synchronization as well as multiple sensing modalities.

3.5.4 Experimental Deployment

We constructed six multimodal sensors and installed the REIP SDK on each as shown in Figure 3.6. The experimental setup consists of a 3 x 2 arrangement of these sensors across a 20 by 10 meters test area (Figure 3.10). As test signals, a loud impulsive sound source was used (hammer strike on the metal box), as well as a sine wave generator, resembling a vehicle reverse alarm.

3.5.4.1 Localization Using Global Synchronization

The first experiment aims to evaluate the ability of the sensor network to localize loud impulsive sound sources (a real-world example of such a sound source is a vehicle impact) using the wireless synchronization technique introduced in Section 3.5.3. To do this, we define Test Route 1 (see Figure 3.10) horizontally

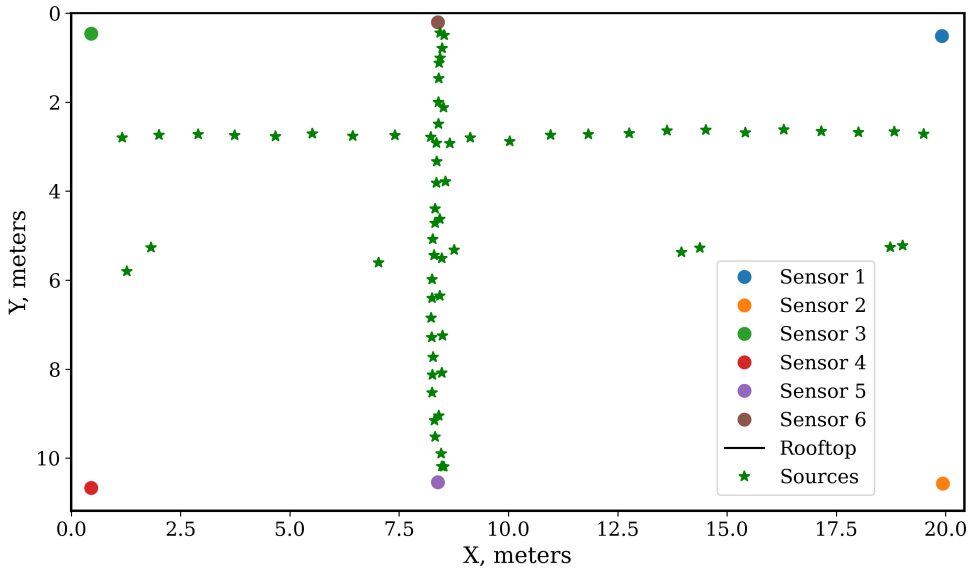


Figure 3.11: Localization of loud impulsive sound sources using global synchronization across multiple sensors. The reconstructed positions are in good agreement with test routes 1 and 2 of the rooftop experiment.

across the field with 3 ft intervals, and Test Route 2 along which the loud impulsive sound is generated with 3 ft intervals from sensor 5 to sensor 6 and with 1.5 ft intervals on the reverse path. Each sensor collects audio and video data continuously and simultaneously with associated timestamps. We also used this hammer source to synchronize the video to audio in a standard way for reference (see Hammer spots on Figure 3.10).

In order to reconstruct the position of this sound source, we first detect the high amplitude peaks t_i in audio data, synchronized using the common time scale as reconstructed by the Timestamp Decoding block (see Figure 3.8). With the known 3D positions p_i of the sensors, one can find the sound source position p at time t by minimizing the errors:

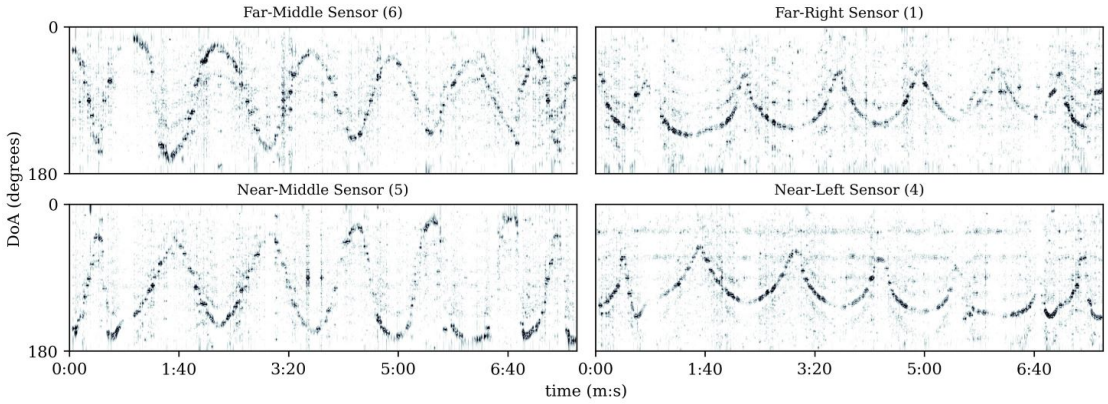


Figure 3.12: Sound source localization results across 4 sensors for the sine wave generator. Sensors 2 and 3 (near-right and far-left corners) are omitted for brevity. For sensors 5 and 6, the back-and-forth path that the sound source was traveling along is very clear as shown by the oscillating path on the corresponding Direction of Arrival (DoA) subplots.

$$p, t = \underset{p,t}{\operatorname{argmin}} \sum_{i=1}^{6} \left(||p - p_i|| - c \cdot |t - t_i| \right)^2, \tag{3.1}$$

where $c = 343\ m/s$ is the speed of sound in air. The results are shown in Figure 3.11 and are in good agreement with the predefined test routes in Figure 3.10.

3.5.4.2 Localization Using Multiple Modalities

During the second experiment, a team member carried a sine wave generator and walked back and forth along Test Route 3 across the span of the test area (see Figure 3.10), stopping periodically to activate a sine wave signal. This sound was chosen because of its simplicity as a sound source and because the acoustic signature strongly resembles that of a large truck's reversing alarm.

Existing sound source localization approaches implemented in the *pyroomacoustics* [55] Python package is used to extract horizontal angle coordinates for these sound sources. For each sensor, the 1D direction of arrival was calculated

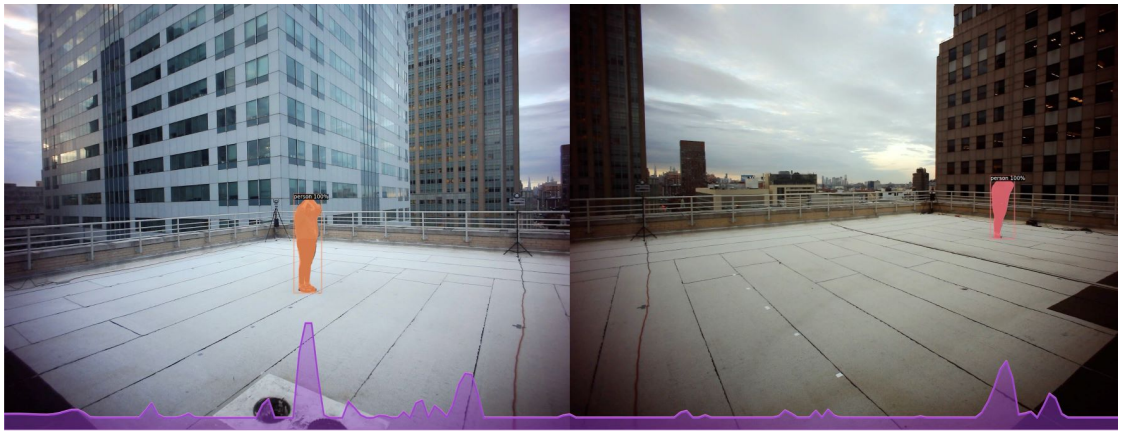


Figure 3.13: Multimodal localization: video object detection and audio source directivity from the perspective of the near-middle sensor (5). Left and right images corresponding to two cameras of the sensor are shown for two different moments in time.

using the center row of each microphone array (4 microphones each spaced 93 mm apart). Only the center row was used to simplify the problem into 2D space and 1D localization as the sound source was held at the same height as the microphones. Figure 3.12 shows the horizontal direction of arrival over time relative to each sensor, arranged according to their physical locations around the testbed.

One can see in Figure 3.12, as indicated by the increase/decrease in the path's amplitude, that the source starts close to sensor 6 and far from sensor 5, and by the end of the experiment it has moved close to sensor 5 and far from sensor 6. This is because, as one moves further away from a measurement point, the angular distance traveled in order to cover the same physical distance is smaller. Similarly, sensors 1 and 4 have a sharper peak and a broader trough because their perspective results in a skewed path, due to the same effect.

Figure 3.13 shows an example of multimodal localization using both video and audio data. Video-based object detection, shown by the orange silhouette, was performed using Mask-RCNN as deployed in Facebook's Detection2 library [56].

The purple area plot at the bottom of the figure shows the predicted location of sound sources aligned with both the left and the right camera views, as calculated by the MUltiple SIgnal Classification (MUSIC) direction-of-arrival (DoA) algorithm [57] from *pyroomacoustics*, and which was chosen because of its balance between computational speed and accuracy. These results show promise in the ability of wireless sensor networks to perform localization independently using multiple modalities and that REIP offers an efficient and effective way of implementing them.

3.6 Discussion and Real-World Applications

REIP facilitates and streamlines the process of environmental sensing deployments, providing researchers of varying experience levels with tools and best practices for designing and building sensor networks. We have built a software framework (an SDK) to make it quicker and easier to prototype and deploy sensor networks and we have shown its use in two different case studies, demonstrating its utility and versatility.

3.6.1 Smart Traffic Event Detection

Practically, the presented smart traffic event detection sensor dramatically reduces data throughput requirements for research projects looking to analyze urban phenomena. The rapid implementation of this case study following REIP's workflow and using the REIP SDK shows promise for its use across diverse applications of sensor networks. With a given set of application requirements, a pipeline can be designed that abstracts away a large amount of technical detail typically required to develop software that could perform this set of complex, interacting operations.

A particularly challenging aspect of real-time sensing is moving data between processes that operate on different schedules. This is commonly addressed with ad-hoc solutions and application-specific code that are not generalizable or re-configurable. When projects are under tight deadlines, it often does not make sense to do more than that – the project just needs code that will perform the required tasks, and re-usability and portability are not the primary motivators. The REIP SDK addresses these common computational problems in a general sense so that when a research problem comes along, an experiment can be up and running as quickly as possible using pre-existing functional blocks. Indeed, we were able to reuse many of the blocks used during evaluation in Section 3.3 to build the data acquisition and processing pipeline of the given case study (Figure 3.5).

Another significant challenge in remote sensing is hardware and software integration, which is subject to a number of constraints including computing resources available, hardware I/O offered, sensing options, inter-process data rates, and available remote connectivity options. With an application pipeline defined using the REIP SDK, this integration process becomes less of a challenge, as the blocks chosen dictate the minimal hardware platform that can support it. The presented process of manually benchmarking possible hardware platforms (Section 3.3.2.2) is not ideal but is a precursor to our planned simulation and optimization tool for a pipeline evaluation stage (Section 7.1), where optimal hardware platforms will be matched to an application pipeline in an automatic way subject to user constraints, such as maximum memory usage, data output rate, etc.

3.6.2 Object Localization and Tracking

The case study in Section 3.5 highlights the use of the REIP SDK to design the application pipeline for multimodal tracking in urban environments. A finding from this case study was that the REIP workflow allows the sensor network architect to design the data pipeline from an abstract, top-down view and have it translate directly into software components without having it get over complicated with software and hardware-specific details. More importantly, it allows the hardware decisions to be pushed to a later stage in the process which, if made too early, can constrain the application unnecessarily. An example of this was an on-demand integration of the more accurate synchronization solution that was required for audio-based localization.

The modular blocks that were combined to make up this case study functionality abstracted a lot of the complexity away from the traditional software development process. On top of that, the implementation stayed clean and easy to maintain through both cases. The use of generalizable serialization methods in REIP SDK for inter-process data exchange has also provided the efficiency necessary for handling multiple sensing modalities at the same time.

3.6.3 Real-World Applications

We are convinced that the modular design of REIP SDK enables a greater reusability of research efforts and leads to faster development of sensor networks. We further explore the benefits of REIP in real-world applications described in the following chapters, which are focusing on: Heat, Ventilation, and Air Conditioning (HVAC) systems, urban traffic analysis, and sports tracking correspondingly. These

application examples are also providing us an opportunity to learn from the experience of REIP users (of different levels of expertise) that implement them, find any shortcomings in our design or implementation, and figure out ways to further improve the system.

Chapter 4

Applications: HVAC Systems

4.1 Motivation

It is well-known that buildings consume around 40% of total US energy use, while heating, ventilation, and air conditioning (HVAC) systems account for 74% of building energy consumption [58], hence improvement in how HVAC systems run in buildings will eliminate the major contributors of energy waste [59]. Current heating, ventilation, and air conditioning practice in buildings assumes a fixed setback schedule as well as over-simplified representation of occupant presence, such as predicted mean vote (PMV), or least enthalpy estimator (LEE) to run the equipment [60, 61, 62]. This assumption typically results in unnecessarily running HVAC equipment at the full capacity to condition spaces without knowing the actual state of spaces or flow of users [63, 64]. Overventilation then results in significant energy use and discomfort for occupants [65]. Given the fact that 84% of the life cycle energy use is associated with the operation phase, inefficiencies brought by such simplifications in energy use are significant [66].

Large opportunities for reducing energy waste exist in buildings if HVAC equipment is run by dynamic reasoning with accurate real-time occupancy and space state information [67, 68]. Current practice fails to consider accurate representations of the current and future state of spaces (in terms of real-time occupancy, user preferences, and comfort levels), and unrealistically assumes that the sensor measurements obtained locally represent the entire building spaces, resulting in waste in the energy, carbon and environmental footprint of buildings. Approaches to reduce energy use in buildings through occupant inputs, such as programmable thermostats, resulted in more energy use than traditional manual thermostats or an uncomfortable indoor environment for the occupants [69, 70]. Similarly, various thermal comfort indices used in the practice of HVAC control have failed to represent a realistic representation of building use patterns. REIP could provide the foundational input for having adaptive control strategies in HVAC systems.

The main objective of this application example is to conduct a comprehensive analysis of four weeks of data collected from sensors placed on the 12th floor of 370 Jay street to show the presence of wasted energy consumed by the HVAC system for spaces that are often underutilized, and to provide practical ways of reducing it. We also demonstrate the feasibility of live detection of indoor spaces occupancy using the REIP platform, which could be used in dynamically controllable HVAC systems for even better performance and energy efficiency.

The structure of the chapter is as follows: Section 4.2 describes the methodology used during the study followed by independent analysis (Section 4.3) based on solely the data acquired with upgraded REIP sensors. We then compare our data to the internal building's HVAC system sensor readings and provide additional analysis in Section 4.4 with improvements suggestions in the discussion (Section 4.5).

4.1.1 Evaluation of REIP's Utility

Another objective of this application example is to evaluate how well REIP provides features such as easily extensible support for multimodal sensing. We are also interested in validating whether the design principles of the REIP SDK (Section 3.1.3) did indeed result in an easy-to-learn API. For this purpose, the application example was introduced as a Capstone project for a group of four Master's students at the Center of Urban Science and Planning (CUSP) at NYU. The students had little to no prior experience in sensor design but have taken basic classes in programming (Python) and data analysis.

The Capstone project at CUSP is a two-semester-long program where a group of students with complementary skills is collaborating with the researchers on a data-focused problem. The unique feature of the proposed project is that the students were acquiring their own data by themselves for subsequent analysis, and this is exactly what REIP is trying to achieve – enable non-expert users to perform their research faster without drowning in technicalities.

Based on the nature of the project, the following features of REIP could be assessed: multimodality, extensibility, and easy API. The idea is to provide students with sensors that lack some of the sensing modalities required by the applications so that they have to use REIP's API to extend the sensors' capabilities.

Because of very limited prior experience in hardware design, the students were meeting with authors (who served as their mentors) twice as often during the first semester of the program. That is weekly in-person meetings for two and a half hours when extending the sensors' capabilities and bi-weekly during the second semester covering the deployment of the sensors for data acquisition and subsequent analysis. Additionally, the students were included in the research group's Slack

Figure 4.1: A four steps approach towards application objectives.

channel where they were provided with additional learning resources and asked questions in between meetings. Therefore, "we" in the context of this application example should be understood as the team of students and their mentors.

On the software side, the REIP SDK was sought to follow four design principles: accessibility, extensibility, multimodality, and scalability. We use this application example to evaluate the adherence of our REIP SDK implementation to the first three of these principles. For that, we focus on students with limited prior experience, and our findings are discussed in Section 4.5.3 at the end of this chapter. The scalability and HW/SW integration are explored as part of the remaining application examples in Chapters 5 and 6.

4.2 Methodology

One of the analysis objectives is to explore the correlation between the performance of the building's HVAC system and the actual occupancy of the indoor spaces. For that, we are going to leverage the REIP platform and a set of prebuilt sensors available at NYU. However, these sensors do not support environmental sensing modality (temperature and humidity in our case) out of the box, so we use

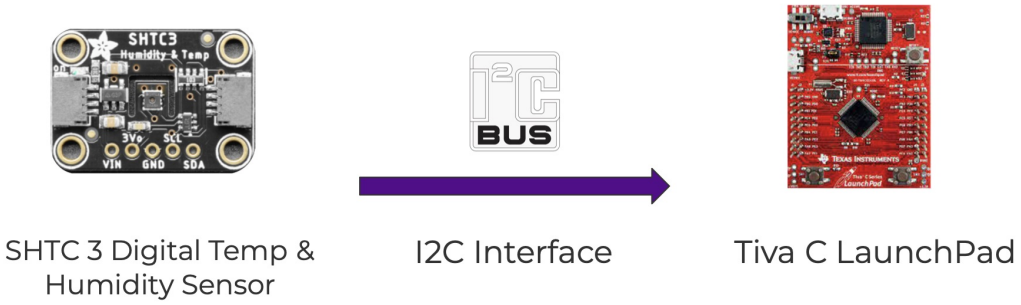
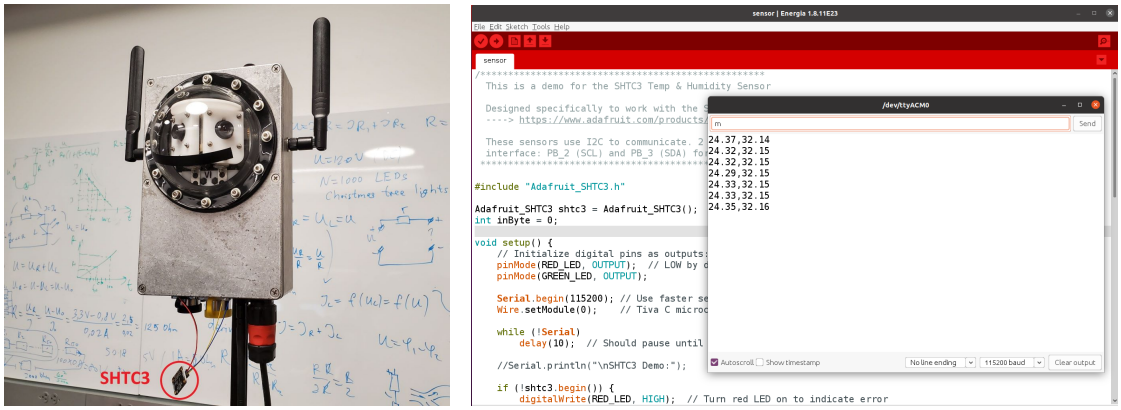


Figure 4.2: REIP sensors' capabilities extension for temperature and humidity measurements using SHTC3 digital sensor and Tiva C microcontroller.

a four-step approach to achieve our objectives (Figure 4.1). First, we extend the sensors' hardware capabilities to support temperature and humidity measurements, which also involves the development of the corresponding new REIP software blocks. Then, we deploy a set of said sensors on the 12th floor of the 370 Jay street building where we can also acquire the internal building's HVAC system data as a reference. Finally, we analyze the data to gain insights into the building's HVAC system performance as well as users' behavioral patterns. More on each step in the following sections.

4.2.1 Sensor Capabilities Extension

To add support for an environmental modality to REIP sensors, we choose the SHTC3 digital temperature and humidity sensor as our sensing device (Figure 4.2). This device has excellent features: high-accuracy (± 0.2 °C) temperature measurements and $\pm 2\%$ relative humidity measurements. Also, it has an I2C (Inter-Integrated Circuit) interface for easy reading. We connected the sensor with the Tiva C Launchpad that was already available in REIP sensors as part of its high-accuracy synchronization solution.



(**a**) Upgraded REIP Sensor (**b**) Microcontroller Firmware (C++)

Figure 4.3: Upgraded REIP sensor (**a**) with corresponding firmware (**b**) developed in Energia IDE [71] for instantaneous temperature and humidity measurements.

The sensor firmware (Figure 4.3b) has been developed in Energia IDE [71], and it provides instantaneous temperature and humidity measurements via serial port connection in response to a short "m" command (stands for measurement).

4.2.2 Sensor Software Development

The default REIP blocks library contains a variety of blocks needed for this application: a video capture block for the cameras, an object detection block with GPU acceleration, a JSON writer, a CSV writer, etc. Figure 4.4 is showing the data acquisition and processing pipeline implementing real-time occupancy estimation by means of a bounding box based object detector for class type "person", and an independent task that is querying SHTC3 for temperature and humidity measurements every second (using *max_rate* feature of REIP blocks) and storing them in a CSV file for later analysis. It is of note that there is only one new REIP block handling SHTC3 sensor readout that needs to be implemented from scratch to build such a data processing pipeline thanks to the extensive REIP blocks library.

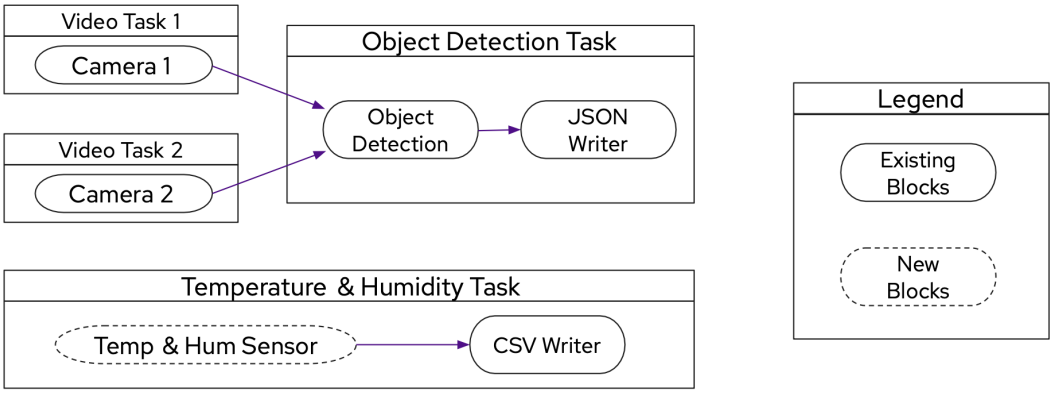


Figure 4.4: Updated REIP pipeline used for data acquisition with upgraded sensors.

For privacy reasons, we do not use audio modality and we also discard any raw video frames after they have been processed by the Object Detection block except a few static images for camera calibration during the installation of the sensors.

4.2.3 Data Acquisition

With REIP sensors upgraded (Figure 4.3a), we proceed to install a couple of them on the 12th floor of 370 Jay Street (CUSP's building). As shown in Figure 4.5, one sensor is located in the Master's lounge and the second one is positioned in the dining area next to the staircase leading to the 13th floor of the building. Sample frames from the first sensor's point of view are shown in Figure 4.6.

The data collection lasted for the duration of one month (June 2022) resulting in a total of 110 GB of raw data recorded by the upgraded REIP sensors. For reference, we have also acquired the building's HVAC system data for the same period. Limited availability of the building's HVAC data (with sensors providing both temperature and humidity measurements) is what has constrained us to such a setup on the 12th floor only of the building.

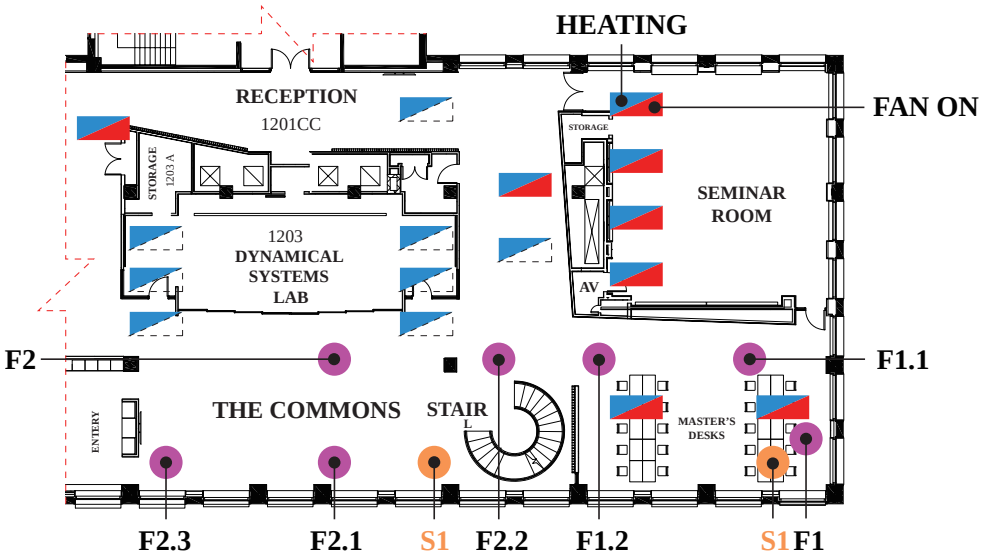


Figure 4.5: Sensors deployment floor plan. Two upgraded REIP sensors S1 and S2 have been deployed on the 12th floor at 370 Jay street: one in the dining areas and one in the master's lounge. Building's HVAC system sensors F2 and F1.1 are being used for data validation (Section 4.4).

4.2.4 Data Analysis

We perform two kinds of data analysis in this application example. First, we investigate the correlation between the building's HVAC system performance and the actual occupancy of the observed indoor spaces (Section 4.3). We then validate our measurements of temperature and humidity against the reference data from the building's HVAC system. (Section 4.4). More details are in the following sections.

4.3 Independent Analysis

Upgraded REIP sensors form a fully self-reliant system and both measure temperature and humidity as well as estimate the occupancy in real-time at their corresponding deployment locations. We achieved an average uptime of ≈80%

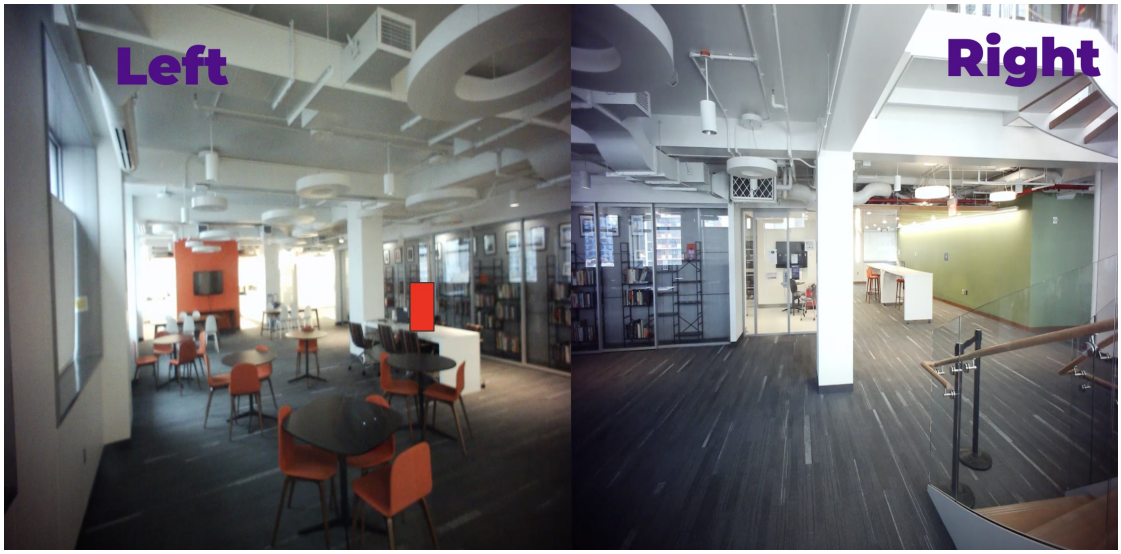
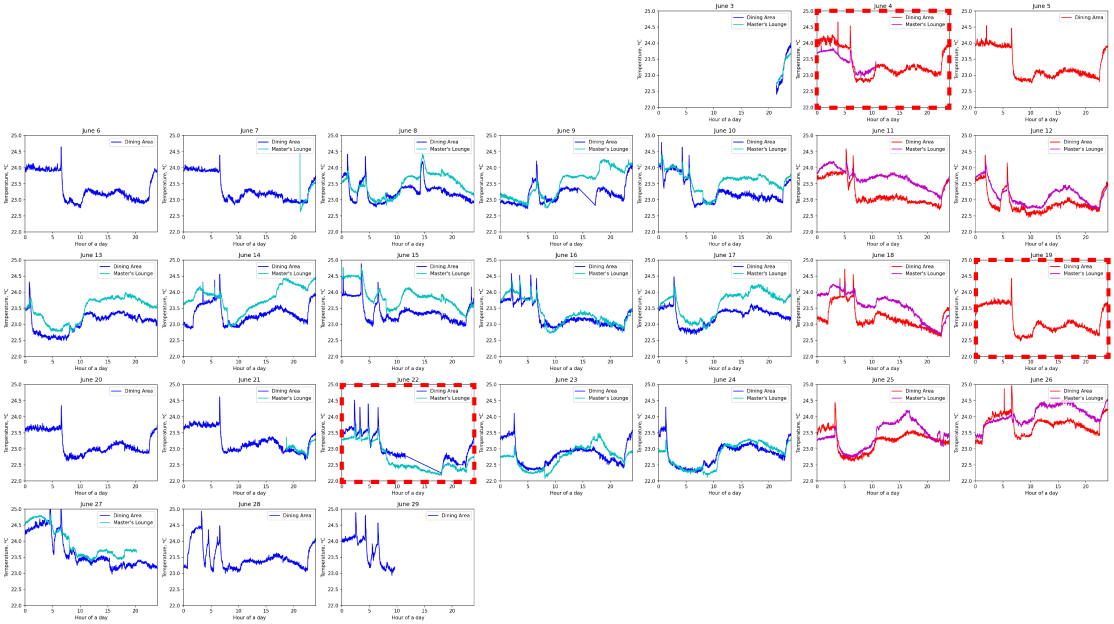


Figure 4.6: Sample frames for the first sensor located next to the staircase in a dining area (with a bounding box of the detected object).

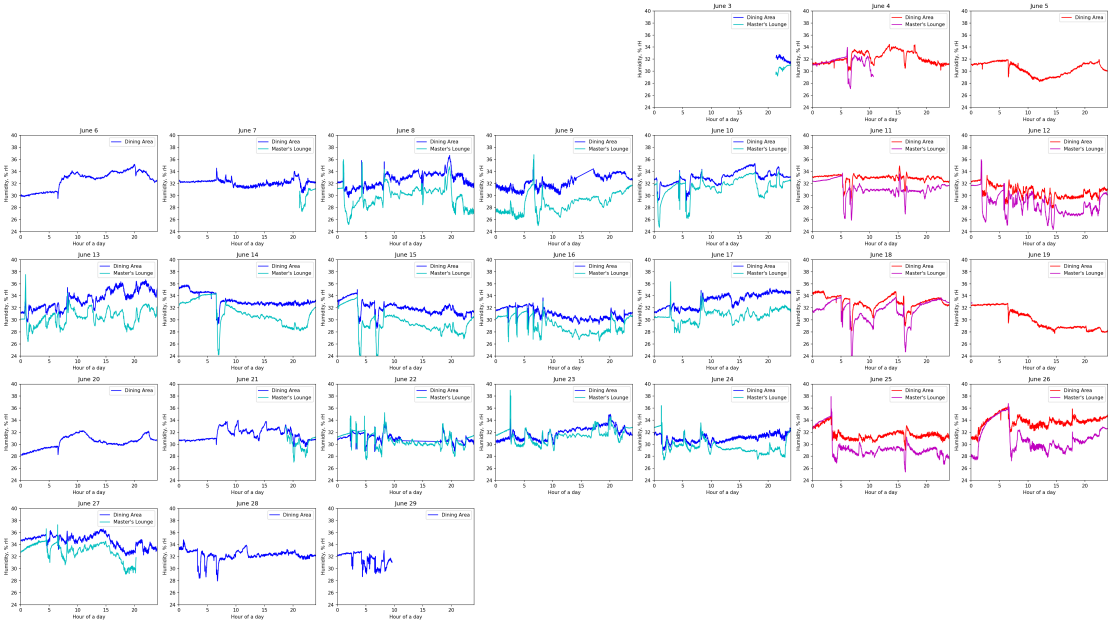
during a month-long data acquisition session (Figure 4.8). Of note is the fact that even if one of the sensing devices has failed, i.e. the left camera due to overheating after prolonged operation, the rest of the data acquisition and processing pipeline could still function correctly in most of the failure cases thanks to the modular API of REIP SDK. We were able to achieve nearly 100% uptime by the end of the data acquisition after ensuring proper camera cooling and reliable electrical connections for the SHTC3 sensors.

4.3.1 Temperature and Humidity

Figure 4.7 is showing the temperature and humidity data acquired by REIP sensors in both the dining area (blue/red color) and the Master's lounge (cyan/magenta color). The data is organized in calendar-like visualization with a constant vertical axis scale for easier comparative analysis. Weekend days are emphasized in red and magenta colors respectively. The dashed red boxes on boxes on Figure 4.7a

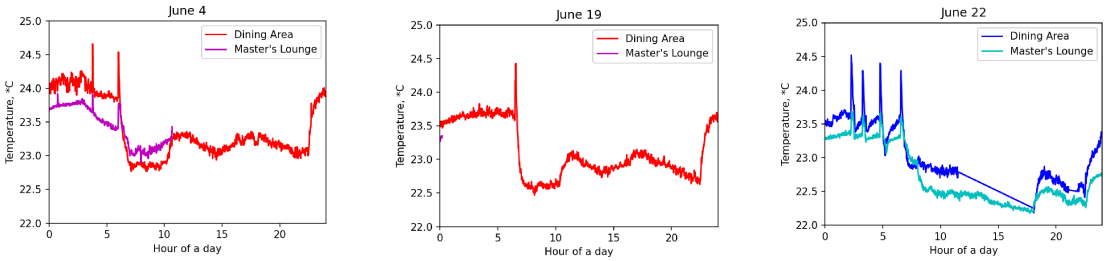


(**a**) Temperature



(**b**) Humidity

Figure 4.7: REIP sensors data for Sensor 1 located in the dining area and Sensor 2 in the Master's lounge during the month of June 2022.



(**a**) June 4 (**b**) June 19 (**c**) June 22

Figure 4.8: Sensors downtime. An average uptime of 80% was achieved for the entire duration of data collection (June 3rd to June 29th, 2022).

are highlighting the failure cases which are shown in more detail in Figure 4.8.

It is apparent from the temperature data that there are two major operation modes of the building's HVAC system, which we will refer to as comfort and economy. The system switched from comfort to economy at 11 PM and back to comfort at 6 PM of the next day. This is an obvious attempt to save energy during the night when the building is largely empty but it assumes a fixed user presence pattern, even throughout the weekend. One can also observe a systematic difference between the temperature measurements from different sensors because of different deployment locations, with Sensor 1 typically reporting slightly cooler temperatures as it was located closer to the air outlet of the HVAC system providing cool conditioned air during the summer month of June. The humidity values largely remain constant both temporally and across different deployment locations.

Let us have a closer look (Figure 4.9) on the day of June 6th for the first sensor located in the dining area next to the staircase between the 12th and 13th floors of the building. It is illustrating a typical pattern observed during many of the other days of the month where the indoor temperature, after initial cool down due to the mode switch at 6 AM, starts to rise slowly as people come to work (or study). It peaks around 11 AM, has a trough around 2-3 PM, peaks again at 5 PM and, finally,

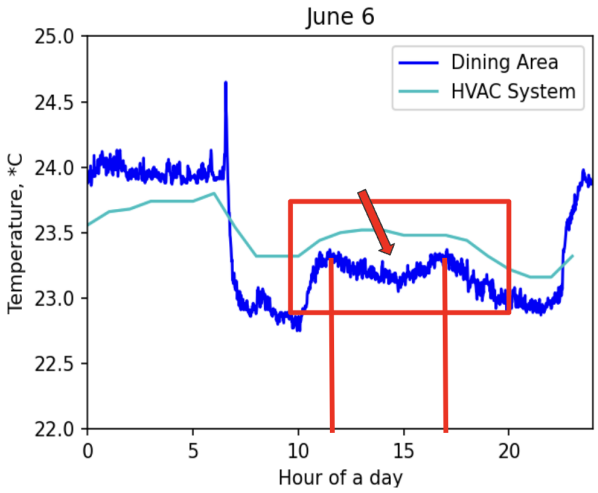


Figure 4.9: Lunch time behavioral pattern. Working at constant power, the building's HVAC systems tend to cool down the spaces a little extra when people are going out for lunch break, which can be seen from REIP's sensors data but not from the building's HVAC data.

gradually declines until the mode change again back to economy late in the evening (at 11 PM). Our hypothesis is that this temperature pattern on a sub-degree scale is due to the students/user activity inside the building during the workday.

4.3.2 Occupancy

Indeed, as shown in Figure 4.10, the period around 2-4 PM is when there is most activity at the staircase as many people are going out for lunch. As they leave, the HVAC system working at constant power managed to cool down the spaces a little extra since every person is an approximately 100 W heater from the point of view of thermodynamics. And when people come back, the temperature returns to its previous equilibrium. Finally, as students and workers gradually depart home after a workday, the temperature is also declining until the HVAC system switched back to economy mode and the temperature spikes up, remaining

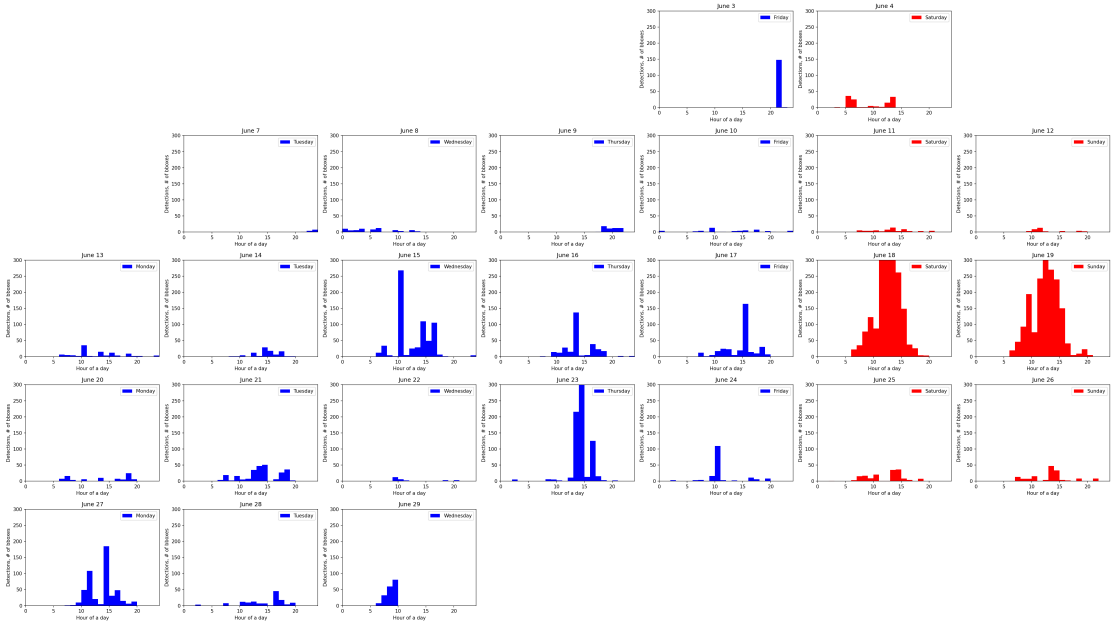


Figure 4.10: Dining area occupancy for an optimal confidence level threshold.

relatively constant for the following night.

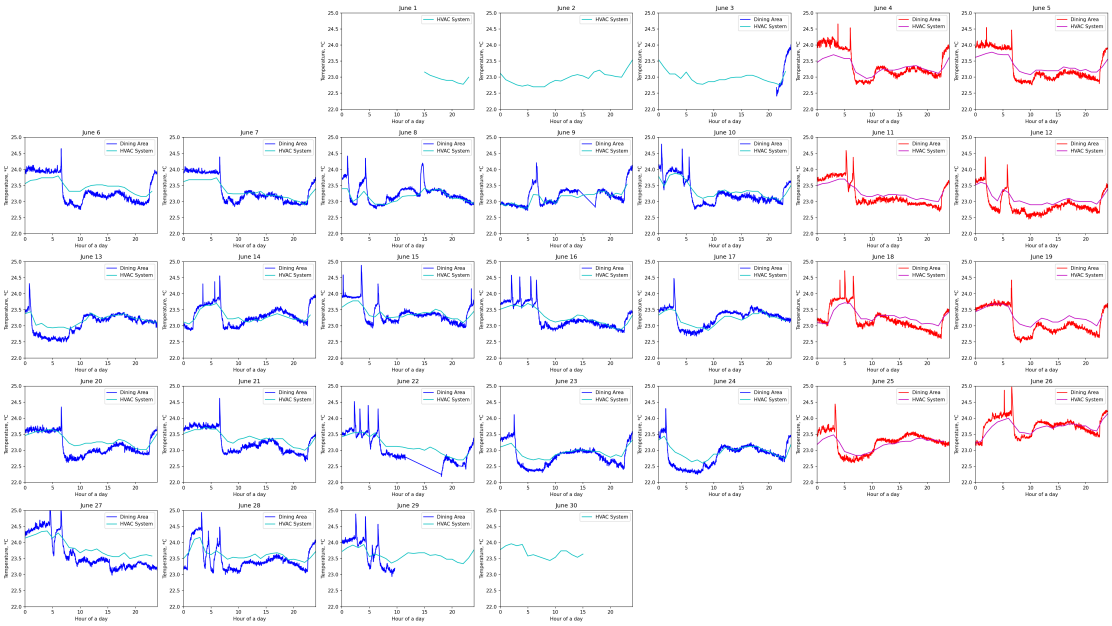
On a technical side, the distributions in Figure 4.10 are computed by averaging the bounding box detections for the class "person" as computed by the Object Detection block in data acquisition and processing pipeline (Figure 4.4) and which is using MobileNet V2 [51] Single Shot Detector (SSD) model for real-time performance. The optimal confidence level threshold was found to be close to 0.5 which results in the highest signal-to-noise ratio (maximum amount of correct detections for the least amount of false positives). The vertical axis is constant across different days for the comparative analysis and an abnormally high activity during June 18-19 is due to the social event happening that weekend. Because the sensor was located next to the staircase, the occupancy data is more representative of the user's activity in the dining area rather than their static presence, which was of benefit for our analysis.

4.4 Building HVAC Comparison

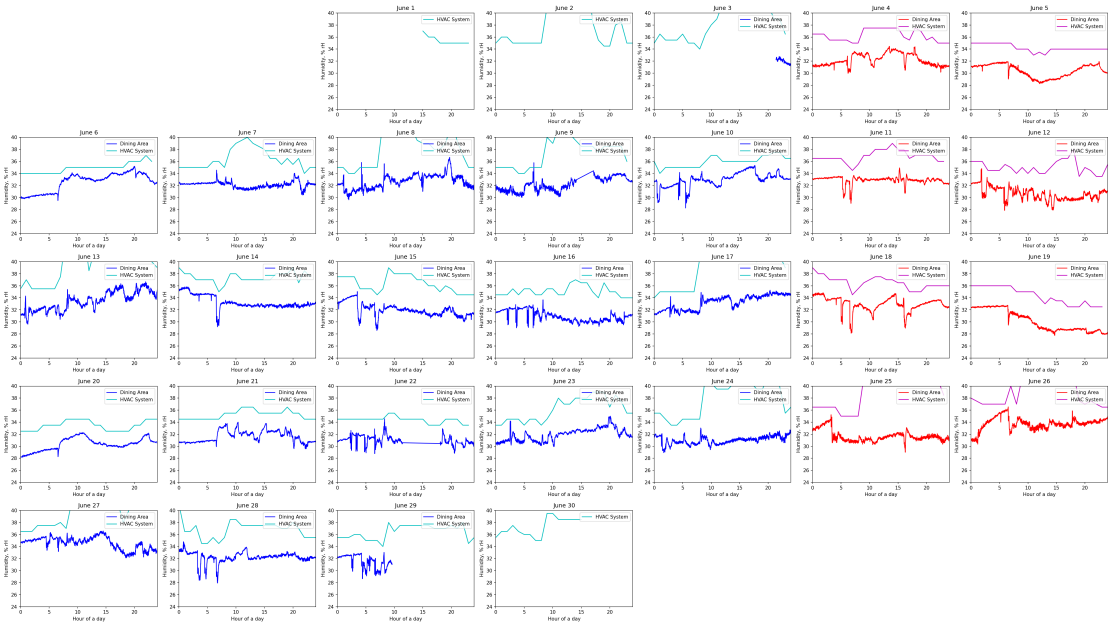
The building's HVAC system consists of a network of mostly temperature-only sensors installed throughout the building floors and Fan Processing Units (FPUs) that use this data to fine-adjust the temperature and humidity of the incoming air, which is being preconditioned by a rooftop portion of the system located on top of the building. Settings of the FPUs indicate that we have correctly identified two main working modes of the system that are switching at 6 AM and 11 PM, and which are defined by different bands of target air temperatures. We further validate the data acquired with REIP sensors by comparing it to the measurements from the nearest temperature and humidity sensors of the building's HVAC system (Figure 4.5) as shown on Figures 4.11 and 4.12.

Temperature and humidity sensors are subject to instrumental and statistical errors. To combat this, measurements are taken every second which can then be averaged to reduce the statistical error and match the one-minute temporal resolution of the building's HVAC data. However, it can easily be seen from Figure 4.11 and the closeup view in Figure 4.9 that there is an extra post-processing step in the HVAC system which produces highly filtered values of lower temporal resolution closer to one hour despite the one minute sampling period. This is likely an attempt to combat any spurious measurements coming from the potentially faulty sensors over long periods of operation but has a negative impact on the accuracy of any analysis done using this data since phenomena such as the users' lunch behavior patterns become simply undetectable. Other than that, our data is matching the building's HVAC data within the instrumental errors of the sensors.

Figure 4.12 is showing what is likely a faulty building's HVAC system sensor

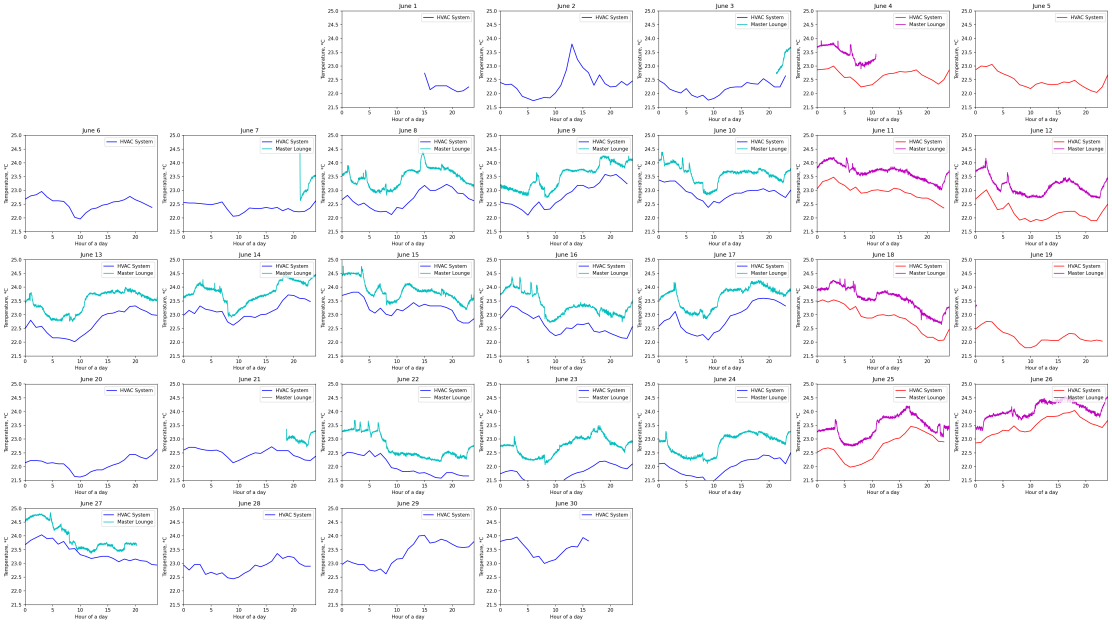


(**a**) Temperature

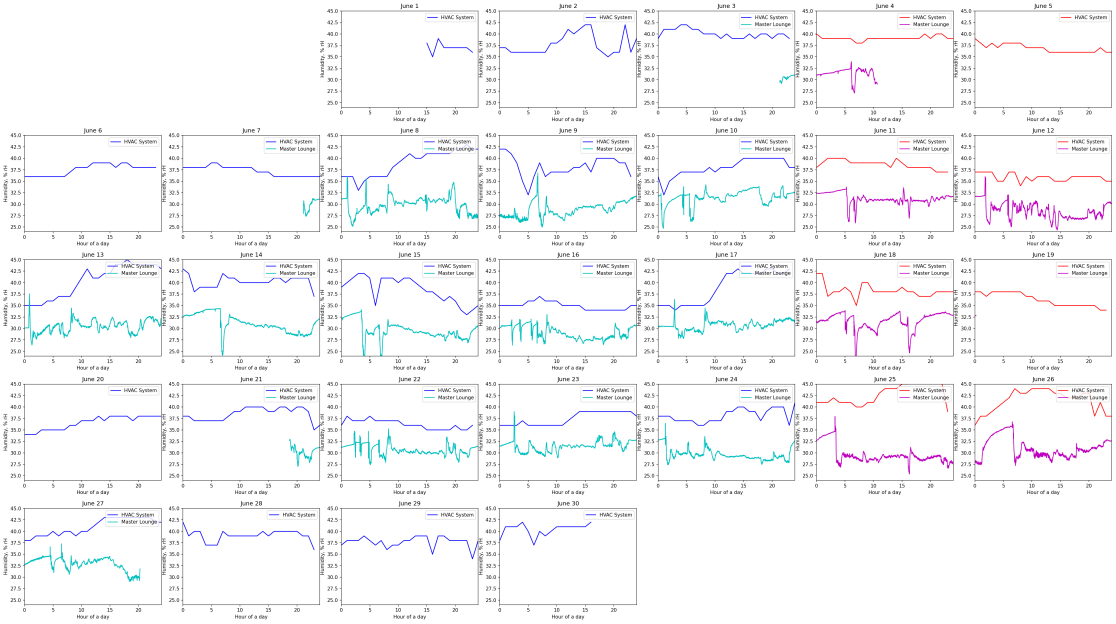


(**b**) Humidity

Figure 4.11: Sensor 1 vs. building HVAC data comparison shows good agreement within instruments errors of both temperature (**a**) and humidity (**b**) measurements.



(**a**) Temperature



(**b**) Humidity

Figure 4.12: Sensor 2 vs. building HVAC data comparison suggest faulty building sensor based on both temperature (**a**) and humidity (**b**) measurements.

F1.1 (Figure 4.5) since the measurements no longer agree with our upgraded REIP sensors which were tested in the lab conditions to work properly prior and after the deployment. There is a systematic bias between the measurements and humidity data coming from the F1.1 sensor which are largely constant. We later validated that the building sensor has indeed failed by creating artificial changes in the air condition next to the sensor. The sensor did not report any increase in the humidity despite the person breathing heavily directly into the sensor, and also application of a heater resulted only in a limited reported temperature increase compared to other sensors. The sensor F1 worked perfectly though and is located closer to the second REIP sensor but we were unable to obtain any data for it from the building's HVAC system export tool because of a mismatch between the digital system configuration and actual physical deployment of the sensors.

4.5 Discussion

In this application example, we acquired using REIP (SDK) accurate temperature and humidity measurements as well as estimated the occupancy of indoor spaces on the 12th floor of the 370 Jay Street building. We provided an independent analysis of the acquired data as well as validated its accuracy against the building's HVAC system readings. Several conclusions can be made about the building's HVAC system performance based on our analysis.

4.5.1 Implications

HVAC System Improvement

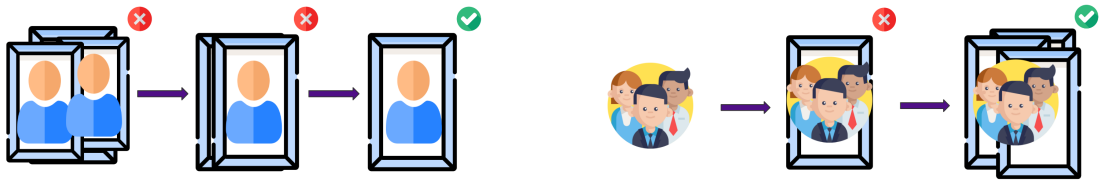
Our analysis indicated that the building's HVAC system is switching to comfort

mode as early as 6 AM of every day and remains in this mode until 11 PM (17 out of 24 hour total in a day). At the same time, the occupancy data suggests that the main user activity in the building is largely limited to the time interval between 9 AM and 8 PM, i.e. 6 hours less per day. This indicates the presence of significant energy waste which, based on our findings, could be reduced by approximately a third of what the HVAC system is consuming extra in comfort mode relative to the economy. One would only need to update the system schedule to better reflect the actual user usage of the building spaces.

Furthermore, even temporary changes in the building occupancy have been shown to impact the resulting temperature levels due to the fact every person represents a low-power heater by virtue of higher body temperature relative to the surroundings. The ability of REIP sensors to provide real-time estimates of the occupancy of the indoor spaces without infringing on users' privacy (video data is being analyzed in real-time and discarded immediately) could be leveraged to form an active control feedback loop for the HVAC system to further optimize its performance. For instance, the power consumed by FPUs could be dynamically adjusted to lower levels when people go out for lunch to not air condition the spaces too much unnecessarily.

NYU and Climate Change

The world is dealing with rapid urbanization, higher demand for energy consumption, and a tremendous increase in greenhouse gases. The primary source of New York University's emissions are the buildings which account for 0.3 % of New York City's greenhouse gases, and 60 % of NYU's buildings are older than 60 years with inefficient HVAC systems [72]. Therefore, New York University has taken climate change action to reduce its greenhouse gas emissions and to reach



(**a**) Overestimation (**b**) Underestimation

Figure 4.13: Occupancy error types: (**a**) overestimation when multiple bounding boxes are reported for the same person; and (**b**) underestimation when there are missing bounding boxes for undetected people.

carbon neutrality by 2040 [73].

This application example has demonstrated the feasibility of using the REIP platform for addressing such challenges and we were able to provide insights into users' behavior with actionable suggestions on how to improve the existing HVAC system schedule to minimize energy waste. 370 Jay Street building, where the Center of Urban Science and Planning (CUSP) is located, is already one of NYU's Leadership in Energy and Environmental Design certified buildings. We believe that REIP has the potential of contributing to this initiative by making the building's HVAC system smarter and more adaptive. Moreover, the user-friendly design of REIP SDK has made it accessible to non-domain experts. The application was proposed as part of the CUSP Capstone project and was accomplished by four Master students with no prior experience in sensor network design and deployment (under the supervision of the authors during the Spring and Summer 2022 semesters).

4.5.2 Limitations and Future Work

Initially, we were aiming to deploy half a dozen sensors on both the 12th and 13th floors of 370 Jay Street occupied by CUSP. However, the building's HVAC system data from the 13th floor is not available for research purposes, so we were

limited to the 12th floor only. Now, the REIP sensors and research methodology have been validated, and we can deploy a larger number of sensors on the 13th floor as well and do independent analysis for a larger number of indoor spaces. Also, although sensor deployment next to the staircase has proven to be useful for the analysis of relative user activity, it is not well suited for static occupancy estimation.

It is clear how to deal with measurement errors for temperature and humidity. Statistical errors can be reduced by repeating the measurement and computing an average value, and the instrumental errors are provided by the manufacturer. However, it is much more difficult to estimate the error of occupancy measurements because these are based on a machine learning model (Figure 4.13). An intersection over union based and temporal filtering can be used to improve the accuracy of occupancy measurements. Calibration of sensors in the environment with controllable occupancy would also be required for absolute measurements and the impact of low lighting conditions should be investigated.

4.5.3 Lessons Learned

Several lessons have been learned about REIP from the interaction with the students and their use of the infrastructure. First is that documentation in a form of complete examples is very effective but also REIP blocks themselves follow a well-thought-through structure, which turns each block into effective documentation on how to perform individual tasks as well as how to use various third-party libraries common for sensor design. The students were able to extend the sensor's capabilities by adapting the existing block for communication with the microcontroller via UART protocol. The most challenge was to develop the firmware for said microcontroller as it required a different programming language (C) and domain

knowledge. That shows that a greater variety of data source blocks has the most potential for reduction of the development time in the future.

A similar observation could be made about the existing data sink blocks which mainly focus on storing the data locally. A major factor in achieving high uptime for the sensors is the ability to monitor the health status of the sensors. We relied on existing remote control protocols such as SSH and VNC for this purpose which resulted in delayed identification of the sensor failures and increased data loss. Additional monitoring blocks of the data sink type that can report sensor status directly into a centralized database connected to the dashboard would help improve the reliability of sensor networks powered by REIP.

On the flip side, the REIP SDK has shown a great degree of modularity so only one custom data source block had to be implemented for the new environmental modality to get the minimal data acquisition pipeline working (Figure 4.4). Furthermore, the concurrency tools offered by REIP SDK, such as Tasks, allowed the sampling of one modality (i.e. the temperature and humidity) to continue operating normally even when the other part of the pipeline failed (i.e. cameras overheated). This is a direct result of the design principles we followed.

Chapter 5

Applications: Urban Dataset

5.1 Motivation

Empowered by the 5G technology and cloud computing, future smart cities can utilize the network of sensors to generate a tremendous amount of data to potentially enhance the life of their citizens. We can envisage scenarios where sensor technologies can in "real-time" sense the environment and use AI techniques to broadcast them as out-of-horizon events to different moving entities, such as autonomous and human-operated automobiles.

In this application example, we would like to perform the first study and generate the first dataset using earlier-developed specialized REIP sensors capable of recording both audio and video data (Figure 3.6). The data will also be synchronized across all sensors with high accuracy for video frames using an adapted version of the custom synchronization solution (Section 3.5.3.2). We will then use computer vision techniques for segmenting and modeling the various moving entities in the urban environment, which could be used for high-level analysis, such as:

1. Counting pedestrians crossing the intersections throughout the study;

2. Object and pose detection of traffic at the intersections;

3. Surveillance-style mosaic rendering for video synchronization debugging.

We are planning to include both crowded and uncrowded intersections to introduce the challenge of occlusion into our detection modeling. To create a balanced experimental setting, where pedestrians of different ages and mobility/vulnerability groups are present, we would like to have a diverse set of intersection locations, such as places next to public schools or hospitals. This is particularly important since many of these groups, such as wheelchair users and people with varying levels and types of disabilities, are absent from large-scale datasets in computer vision and robotics, creating a strong barrier and bias for developing accessibility-aware autonomous systems [74].

The data can then be used in pedestrian-vehicle interaction modeling, pedestrian attributes' recognition models, accessibility-aware autonomous systems, Vision-Zero initiatives, and beyond [75]. Further analysis of this data can also shed light on the type of infrastructure needed to record and analyze such events. One can examine different computational variations, such as whether all four sensors are needed or maybe less can be used to capture the same information, and whether the data needs to be uploaded to the server to further processing or it could be analyzed in real-time by the sensors with edge computing capabilities.

5.1.1 Sensor Options

We are aiming to acquire a novel multimodal (including at least audio and video) and multi-view dataset. There exist several commercially available devices

that seem to be a good fit for such a task so we would like to explore whether that is indeed the case.

A multi-view requirement of our data collection could easily be satisfied with off-the-shelf video surveillance systems that often include a set of wireless IP cameras. These cameras are transmitting their video feeds to a central data storage in form of a local hard drive which sometimes can be synchronized with a cloud but that is not required for the system's operation. The cameras do also include a night mode which can prove beneficial during low-light conditions. However, these cameras rarely provide audio because of privacy concerns and rely on manually configured timing information or NTP (Network Time Protocol) for timestamping of the video. The latter is a significant barrier to a multi-view analysis of such fast-moving objects as car traffic. A car traveling at 40 mph covers more than a meter of ground per frame when recorded at 15 fps. So frame-accurate video synchronization is also a requirement for our dataset and, unfortunately, cannot be met with off-the-shelf security cameras which are also often operating at reduced frame rates out of limited storage considerations.

Another commercial device that provides quality video with audio at a reasonable price is a GoPro camera. However, it was designed for independent operation so does not feature quality synchronization across multiple cameras. Moreover, the synchronization across video and audio modalities is also known to be a problem because of audio lag offset and differences in sampling frequencies. Recently, GPS-based time coding has been introduced in the latest versions of GoPro cameras. That could help with synchronizing the start of the recordings but does not solve the ultimate problem of long-term synchronization. The time drift caused by manufacturing variations of the internal crystal oscillator's frequency that drives digital logic (in-

cluding the sampling frequency) is also susceptible to temperature-based variations. Also, there is no way to know when the GoPro is experiencing lost frames during the recording which ruins the single timestamp-based synchronization altogether. The solution would be a continuous (re)synchronization of the cameras from a single clock source during the entire recording process. Other potential issues include remote control and monitoring of the camera's status as well as weatherproofing that might require external devices and housing depending on the camera version.

REIP sensors provide high-resolution video and audio recording with an in-built synchronization solution. We have demonstrated during the object localization and tracking case study in Section 3.5 the ability to synchronize audio with a single sample accuracy at 48 kHz sampling rate. This made possible sound source localization based on time delays of audio wave propagation in the air. REIP sensors have also been built for outdoor deployments so they include the weatherproofing necessary and external power support. We will be building on top of the existing audio synchronization solution of REIP sensors to achieve frame-accurate video synchronization across sensors as well as across both audio and video modalities.

5.2 Data Acquisition

We've selected three intersections with different demographics of pedestrians and road configurations for the purpose of this dataset acquisition:

1. *Commodore Barry Park*. This intersection is adjacent to a public school. It has a low-to-medium frequency of traffic making it an uncrowded intersection.

2. *MetroTech Center*. This intersection is adjacent to the Chase Bank office building. It is also an active pedestrian intersection.

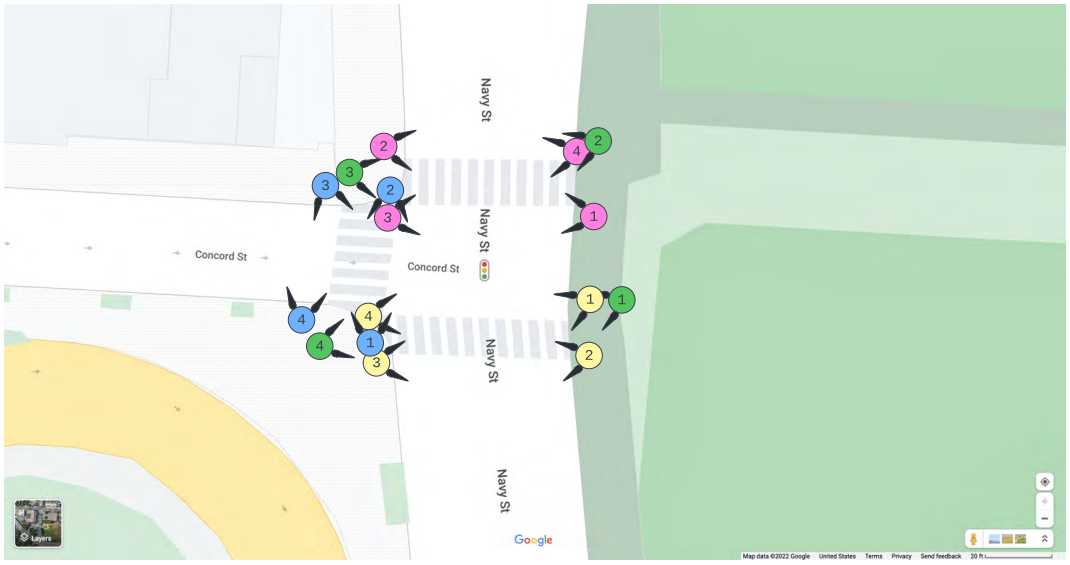


Figure 5.1: Sensor positions during data acquisition at Commodore Barry Park.

Figure 5.2: Sensor positions during data acquisition at MetroTech Center.

3. *Brooklyn Dumbo.* Being a tourist destination, this intersection is the busiest of the three. Because of smaller crosswalks and busy traffic, it provides challenges such as occlusion and a diverse range of pedestrian types.

Figure 5.3: Sensor positions during data acquisition at Brooklyn Dumbo.

We equipped these intersections with four REIP sensors, each placed at one side of the intersection, and record the movement of the pedestrian and vehicle interactions. We recorded four 30-45 minutes long sessions at every intersection. This results in about 200 GB of audio-visual data recorded by each sensor per location (limited by the sensor's max storage capacity of 250 GB). Overall, we collected just over 2 TB of quality data for further analysis.

The detailed map locations and the sensors' positions for each recording session are shown in Figures 5.1, 5.2, and 5.3. Colors denote different recording sessions and the number indicates the sensor number with black arrows pointing in the direction of the field of view of its cameras. Each sensor is equipped with two 5 MP USB cameras providing a 160° horizontal field of view (85° max per camera) at 15 fps recording rate. The 4 by 3 microphone arrays of each sensor are recording at 48 kHz sampling rate. Every sensor was powered by a portable power station with 300 Wh capacity and we also included an Ouster OS-1 LIDAR sensor as an

experimental data source.

This and other application examples have a dual purpose of also evaluating the usability of REIP by users of various backgrounds and technical skill sets. In this particular appplication, the team consisted of two Computer Science Master's students working under supervision of researchers in the VIDA lab. The students did undergo an initial week-long training on how to use the system and were supervised during the first deployment of the sensors at the intersection. Because there was no need in extending the sensors' hardware capabilities, the focus of this application example as a user study was on the students' ability to adapt/use the existing REIP tools during the data acquisition stage as well as use the additional metadata provided by REIP SDK (e.g. the synchronization timestamps) for the subsequent data analysis.

The data acquisition pipeline for the sensors fully consists of the blocks already available in the REIP SDK. We only modified the default Camera block to enable more accurate video synchronization also in presence of significant amounts of lost frames. The reason is that the data acquisition was conducted during the hottest summer month of August, which resulted in the throttling of the sensor's computing platform NVIDIA Jetson Nano after prolonged recording during extreme temperature conditions.

Figure 5.4 is showing the internal timing diagram of the data acquisition pipeline. There are three sources of time in the system: (1) the camera's driver as part of GStreamer (tgstreamer), (2) system time available via time.time() function in Python (tpython), (3) and global timestamp received by radio module at 1200 Hz rate as part of custom synchronization solution (tglobal). Every image frame that was not lost is timestamped using all three time sources. We describe in the next

section how we process this data to achieve frame-accurate video synchronization across multiple sensors with minimal jitter.

Overall, REIP sensors have demonstrated great versatility in data acquisition pipelines and operating conditions. They have even survived without damage a sudden rain incident during the data recording session at Commodore Barry Park.

5.3 Data Processing

For each recording session, the sensors produce two kinds of data: (i) a sequence of 5 seconds chunks of 16-channel audio data, and (ii) a sequence of ≈1 minute chunks of stereo video data with the accompanying timestamp's metadata. We have already discussed in Chapter 3 how the serialized timestamp signal embedded into audio data as a separate channel can be used to synchronize the audio streams from different sensors with a single sample accuracy (Section 3.5.3.2). In this section, we will focus on how one can achieve a frame-accurate synchronization of the video data as well across multiple sensors for free-running cameras, that is cameras take pictures independently at their own pace.

In the data acquisition phase, we timestamped each frame with GStreamer, Python, and Global timestamp (Figure 5.4). All of these clocks have different temporal resolutions and jitter. Figure 5.5 illustrates how equally spaced video frames captured by the camera (tcamera) accumulate time delay and jitter throughout the data acquisition pipeline. Figure 5.6 is showing a quantitative example of this progression for the left camera of sensor 1. Of note is the fact that Global timestamp has the largest jitter due to the communication delays in Bulk_USB block.

Unfortunately, there is no metadata provided by the camera driver that could

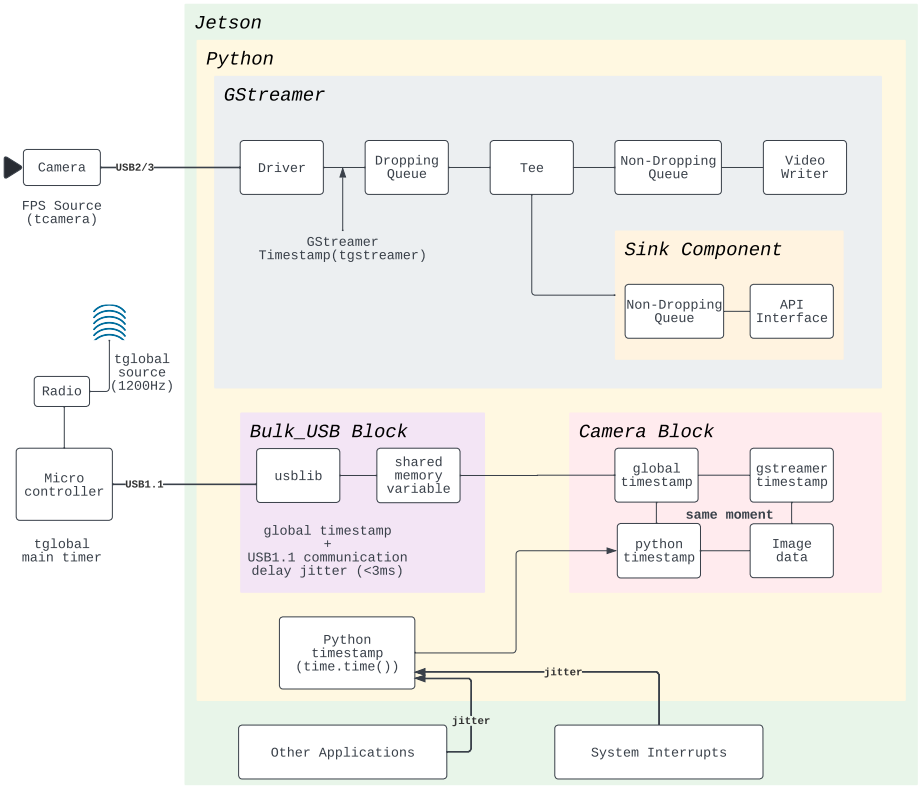


Figure 5.4: Sensor's internal timing diagram for video synchronization. Each video frame is timestamped three times: (1) by the camera driver inside the GStreamer pipeline, (2) by the operating system inside the REIP Python pipeline, and (3) by the radio module inside Microcontroller.

contain clean timing information for each frame, so the GStreamer timestamp is the best we have. Moreover, it is different for every sensor and even each camera with one sensor (in terms of time origin) because left and right cameras are being served by separate REIP blocks, each with its own GStreamer instance that starts with a slight delay one after another. And one should also account for the fact that independent clocks would drift with respect to each other even if they started at the same time because of manufacturing errors of the underlying hardware crystal oscillator.

However, the jitter of GStreamer timestamps is low enough to reliably detect whenever there is an instance of lost frames (and how many) to recover the

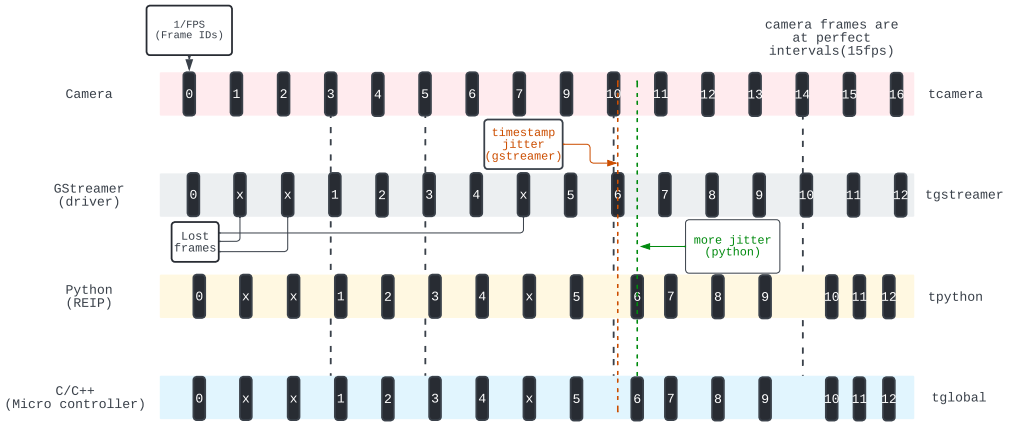


Figure 5.5: Jitter is introduced into the timestamps at different levels of the data acquisition pipeline, by either system interrupts or communication delays.

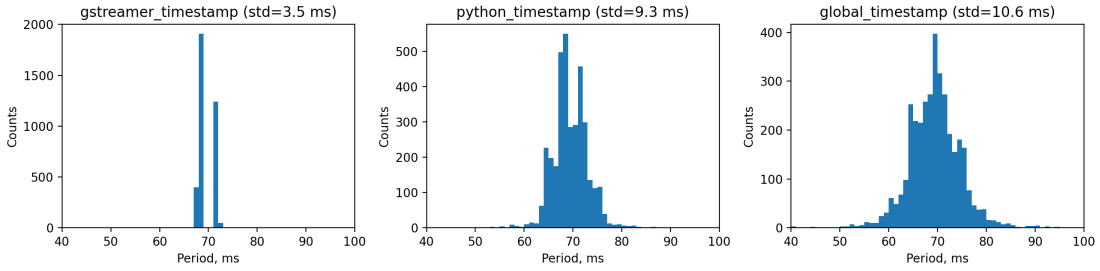


Figure 5.6: Example of jitter progression throughout the data acquisition pipeline.

original video timeline. Therefore, we start with this refined video timeline in terms of GStreamer timestamps and perform two conversions. First, we transform GStreamer timestamps into Python timestamps using a regression line between the two which was computed using all the available timestamps data (Figure 5.7, top-left). Then, we repeat the same step but now between Python timestamps and Global timestamps (Figure 5.7, top-right). This way we are not only bringing all the cameras into a single Global time domain but also eliminating the issue of clock drift because Global timestamps are based on the 1200 Hz clock transmitted by a master radio to each sensor and the sensors are constantly phase-adjusting their

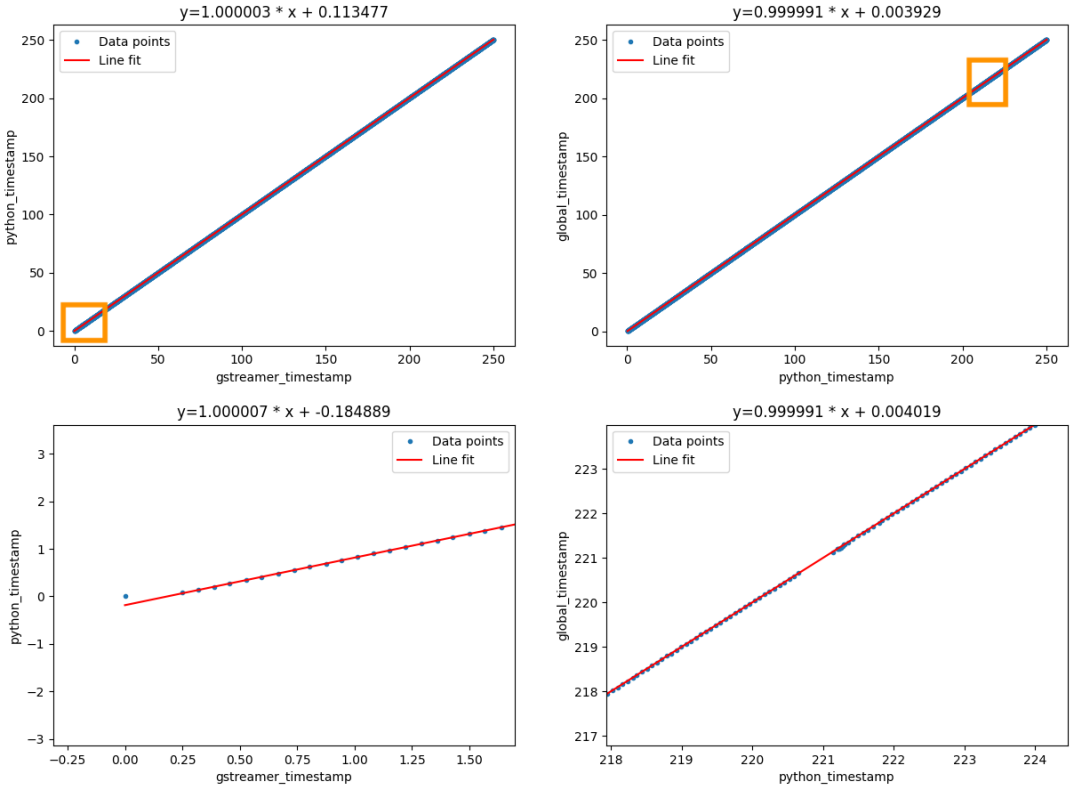


Figure 5.7: Various timestamp artifacts can be corrected during post-processing, including gaps due to the lost frames (left) and queue overflows (right).

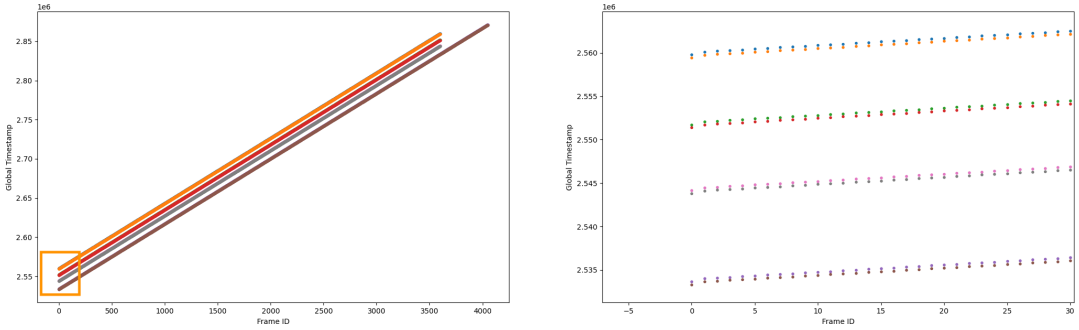


Figure 5.8: After correction of the timestamps, a complete timeline of the recording session can be reconstructed using a single (global) time axis for all sensors.

local Global clocks inside of the microcontroller. We also eliminate the jitter with this synchronization procedure.

Figure 5.9: Mosaic rendering of the synchronized frames from the Chase Bank crossing recording session. Four sensors with two cameras each provide eight different views for comprehensive analysis of the intersection.

As a bonus, we are not only correcting for the lost frames as part of the synchronization process (Figure 5.7, bottom-left) but also fixing any queue overflow issues that often result in jamming of multiple frames after a big time delay (Figure 5.7, bottom-right) once the source of delay (i.e an operating system interrupt) is resolved. The same applies to "frozen" Global timestamps when Bulk_USB block is interrupted.

Figure 5.8 is showing the results after applying the above-described synchronization procedure. One can clearly see how all sensors started recording with a delay one after another because they were started manually in sequential order. One can also see on a zoomed plot the internal delay between the startup of two cameras and a couple of frames that are always lost due to the extra initialization have also been accounted for properly.

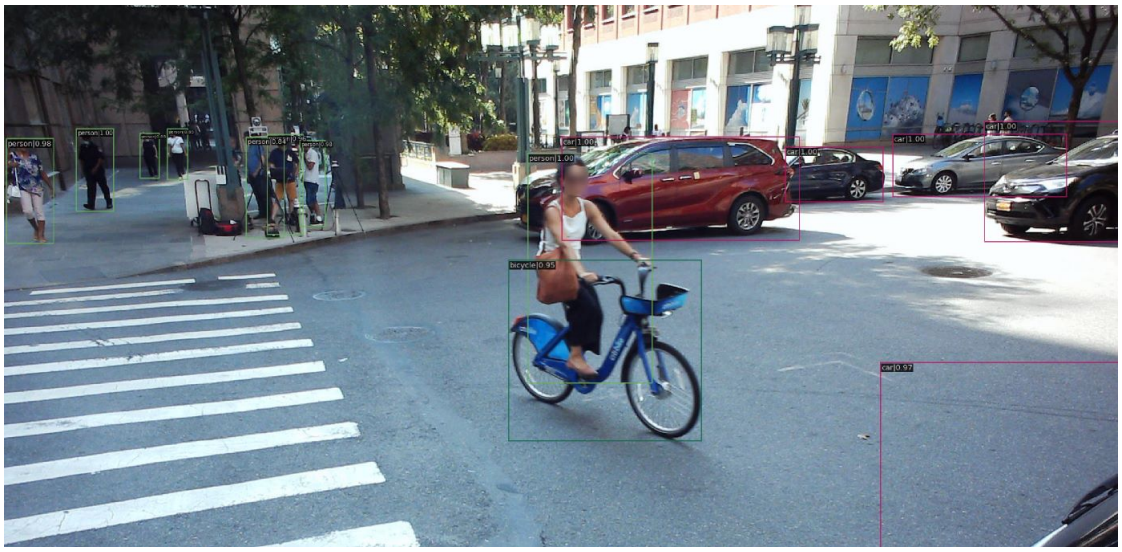
To further validate the video synchronization, we render a surveillance-style mosaic video for debugging before we head toward the analysis of the data. The 4K mosaic video is rendered using processed frames from eight cameras and a global timeline produced by the synchronization of timestamps. Figure 5.9 is showing a sample frame from such a video generated for an intersection next to the Chase bank. At any given moment, the recording of all traffic traveling at substantial speed remains in sync from multiple viewpoints which are essential for many data analysis applications.

5.4 Data Analysis

As a sample data analysis, we apply computer vision techniques to perform object and pose detection on our data. We utilize HRNet – a state-of-the-art bottom-up human pose estimation method for learning scale-aware representations using high-resolution feature pyramids [76]. The network is trained on COCO dataset and the results can be seen in Figure 5.10.

To perform object detection and counting we utilize the HRNet model which is adapted from the Faster RCNN network. We provide detections for six classes: person, car, bicycle, truck, motorcycle, and bus. Figure 5.10a is showing the bounding box visualization of the detected objects. The model has proven to perform well on our data in challenging situations such as a significant change in scale and occlusions.

For pose estimation, the model is executed for each detection of the person independently with a focus on that particular bounding box detection. Such an approach results in temporally consistent pose estimation as the person is walking



(**a**) Object Detection

(**b**) Pose Detection

Figure 5.10: Examples of possible AI inference for sample frames from the dataset.

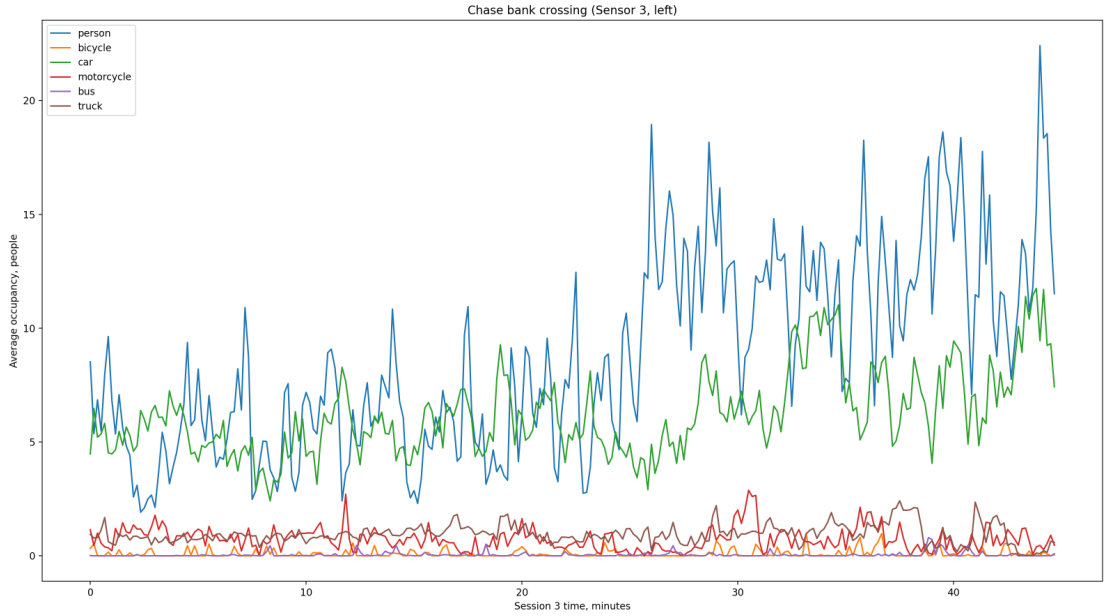


Figure 5.11: Chase Bank crossing occupancy per object/vehicle type during the third recording session in the afternoon. There is a significant (about 3x) increase in the pedestrian count (blue) around 5 pm when people are going home from work.

towards or away from the camera, also despite the significant lens vignetting and brightness variation across the image (Figure 5.10b).

Figure 5.11 is showing the total count of different kinds of traffic participants throughout the entire recording session at the intersection next to the Chase bank office. We intentionally choose this particular recording session because it was conducted around 5 p.m. in the afternoon when people are finishing their workday and leaving home. That results in a spike in pedestrian and car traffic and we aim to detect this event in our data by means of occupancy analysis. And indeed, there is a nearly three times increase in the pedestrian count crossing the street around this time which inversely correlates with car count because cars need to yield their rights to pedestrians. We do not observe as big of an increase in cars or other motorized vehicles count though because this intersection is typically busy

during the normal hours of the day too and there is limited space on the driveways to fit more cars compared to the pedestrians on the sidewalks. Also, parked cars introduce a certain static background count.

5.5 Discussion

We managed to collect unique data about traffic and pedestrians from three intersections using customized REIP sensors. The data includes multiple modalities (audio, video, and LIDAR) with highly accurate temporal information. We used state-of-the-art machine learning models for sample analysis of the video data and the results matched expectations, proving the dataset useful for this kind of task.

At closer inspection, one can notice in Figure 5.11 a regular pattern in the bus occupancy as there is indeed a bus route passing through the intersection. This further highlights the importance of our synchronization technique with diligent processing of timing information to correct for any lost frames that might accumulate into a significant time gap in the video footage. Otherwise, any attempts at temporal analysis, such as, for example, reconstruction of the bus schedule, would suffer from a systematic error.

The majority of the sensor's hardware is enclosed within an aluminum weatherproof housing. There are heat sinks deployed to offer resistance to extreme temperatures and provide better performance. However, we experienced occasional bursts of lost frames even during operation in shadows due to the random operating system interrupts, etc. Therefore, any long-term deployments would inevitably need to account for these issues in a comprehensive way, which is a challenging task and REIP SDK offers a solution for this.

5.5.1 Future Work

We are currently working on a more comprehensive analysis of the dataset using all of the available data. Our objective is to provide examples of analysis that would not otherwise be possible without the unique qualities of our dataset, such as multimodality, multiview and high resolution with precise synchronization.

One example of such analysis is the evaluation of pedestrians' movement per traffic light cycle. We leverage the multiview and synchronization features of the dataset to reconstruct the timing of traffic lights as seen from different sensor locations/cameras. This will enable us to identify jaywalkers and compute their walking speed for further analysis. The audio modality will then be used to detect any near-accident situations caused by jaywalkers by means of detecting car horn sounds in addition to measuring the physical distance between the cars and pedestrians.

The data will also be anonymized to address privacy concerns and ensure "intelligence without surveillance".

Chapter 6

Applications: Sports Tracking

In this chapter, a modular tracking system is presented comprising a network of independent tracking units accompanied by a LIDAR sensor. Tracking units are combining panoramic and zoomed cameras to imitate the working principle of the human eye, and markerless computer vision algorithms are executed directly on the units. Inference from different sensors and modalities can then be fused together to reconstruct higher-level game events or full skeleton representation for each player.

The structure of the chapter is as follows: Section 6.1 provides the motivation for this application example and Section 6.2 details the proposed system design. We then go through three iterations of the tracking unit prototypes in Sections 6.3, 6.4, and 6.5 respectively. Each section provides examples of data samples or analyses. Finally, Section 6.6 contains a discussion and concluding remarks.

6.1 Motivation

Team sports are an example of complex and dynamic physical environments, real-time tracking and understanding of which has been of great interest to coaches

Figure 6.1: The playing field of a baseball game presents an example of a physical environment that is of great interest for real-time tracking. Yet, it poses a great challenge because of the field scale and complexity of game events.

and spectators alike [77]. Statistical analysis of tracking information helps coaches to plan strategies and to evaluate players, whilst real-time detection of players and the ball provides deeper insight into the game during real-life sporting events and enhances the viewing experience through the display of various statistics such as the speed of the ball or overlay of graphical trails during broadcasting [78].

A variety of tracking systems are currently available [79, 80]. The ball is typically being tracked using radar technologies and player tracking systems can be categorized as active or passive. Active tracking systems employ various kinds of markers/ transceivers embedded into sports apparel (e.g. RFID) and might interfere with athlete's performance [81]. In contrast, passive tracking systems are typically using cameras to passively observe players but lack the speed and/or resolution to be able to track the ball or provide a representation of the player at high level of detail.

Another unmet need is to provide a description of high-level game events which is currently done primarily by hand [10]. This makes such systems very labor-intensive and leads to inaccuracies due to human error. However, recent advances in deep learning and computer vision algorithms enabled markerless detection of

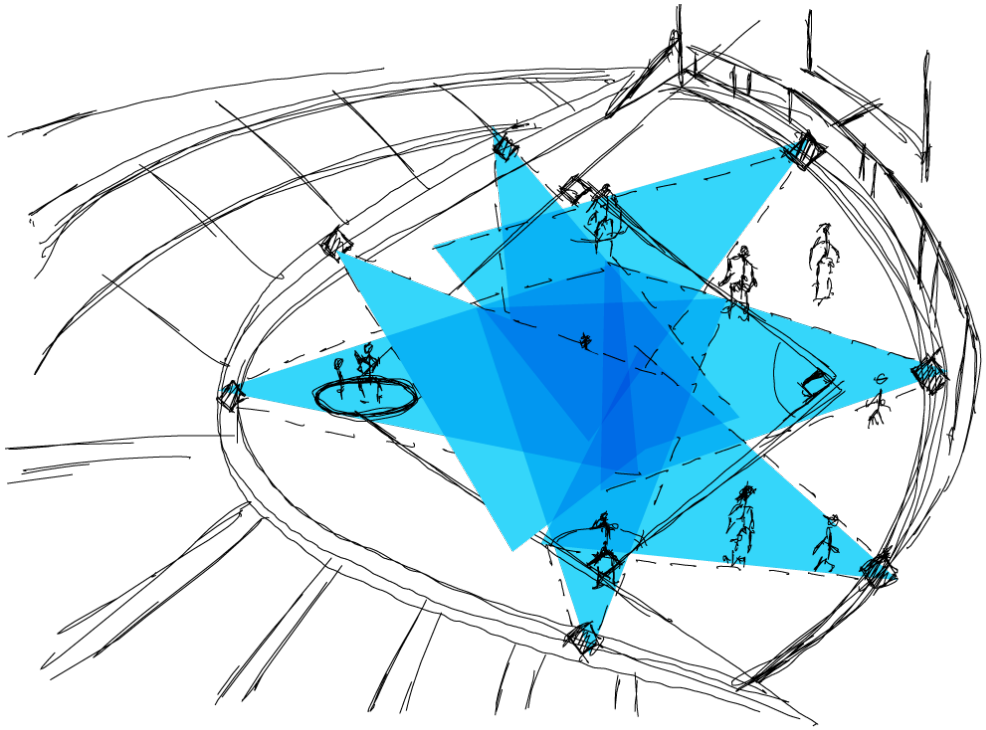


Figure 6.2: LegoTracker Concept. A network of independent sensors that work together to reconstruct a sports game. Running state-of-the-art algorithms for player detection and tracking, they provide a full skeleton representation for each player over a large game field and high-level game events with precise timing.

human pose and action recognition [82, 83, 84, 85].

A typical sports game like soccer or baseball takes place on a field of ≈ 100 meters in length (Figure 6.1). Simply putting a camera overviewing the entire field limits the spatial resolution to the order of several centimeters. Furthermore, most of the pixels and bandwidth are being wasted because players occupy a small portion of the field. A network of cameras focusing on individual players would not have this problem but requires a real-time inference of game events to follow the players. We employ the REIP framework and its workflow to jointly design the hardware and software components of a novel sports tracking system (Figure 6.2) and avoid software challenges by making adjustments to hardware design.

We designed our sensors to be closely focused on a single player. This maximizes the signal-to-noise ratio which we define as the fraction of pixels corresponding to a player over the total image resolution. There is also bandwidth available to use higher frame rates for tracking fast game events, such as a baseball pitch or a bat swing. We are placing tracking units on the ground plane of the game field, and zoomed cameras are motorized in a horizontal direction to follow the players. We are using the NVIDIA Jetson as a computing platform for real-time player detection directly on each unit. This minimizes the latency in the feedback loop of the camera and simplifies the infrastructure needed to operate the system, making it accessible at a lower cost. The units are also precisely synchronized (with 1 μs accuracy) using REIP's novel wireless synchronization solution (Section 6.2.3).

A more in-depth description of the system is presented in the following section, with the results of field tests of different versions of the tracking unit prototypes presented in subsequent sections.

6.1.1 Relevance of REIP SDK

This last application example is the most demanding of the three in terms of real-time performance required for the desired operation of the tracking units. Therefore, we see it as the perfect opportunity to evaluate the scalability of REIP (SDK). It was shown during benchmarking evaluation (Section 3.3) that the REIP SDK introduces limited overhead stemming from data serialization. It is inevitable as part of data management that REIP SDK provides between connected blocks of the pipeline so that users can focus on the desired computations that they perform inside of said blocks. However, these synthetic scenarios had the limitation of using identical cameras that operate at the same frame rate (Section 3.3.1.4). This

application provides the opportunity to test REIP SDK with different cameras, including high-speed ones with higher bandwidth requirements.

The application is implemented by expert users (the authors) who design and build their own hardware units, and leverage REIP SDK with its special features, such as Tasks (Section 3.2.3), to achieve the data acquisition and processing necessary in real-time. Therefore, it is suitable for the evaluation of REIP's HW/SW integration in a hardware-agnostic scenario. Additionally, this application demonstrates the usability of REIP SDK on standard laptop computers with Linux operating systems when operating LIDAR sensor via network interface with a high rate of incoming data packets.

6.2 System Design

Figure 6.3 shows a flat ground field 100 and players 101A-D that are playing within the game field 110. Components of the system are placed around the game field 110. Devices comprising a pair of panoramic and zoomed cameras are referred to as tracking units 200A-D. Dashed lines 211 are denoting the field of view of panoramic cameras and solid lines 221 are showing a momentary field of view of zoomed cameras as it might change over time. The field of view of the panoramic cameras can be greater than 180 degrees to ease the calibration of tracking units. One way of building such a panoramic camera is to combine (and synchronize) several standard cameras into a horizontal array, thus, increasing the horizontal field of view while the vertical field of view remains unchanged. Panoramic cameras of the tracking units 200A-D are sufficiently covering the game field 110 to avoid any blind spots. The system is also complemented with one or more cameras 120

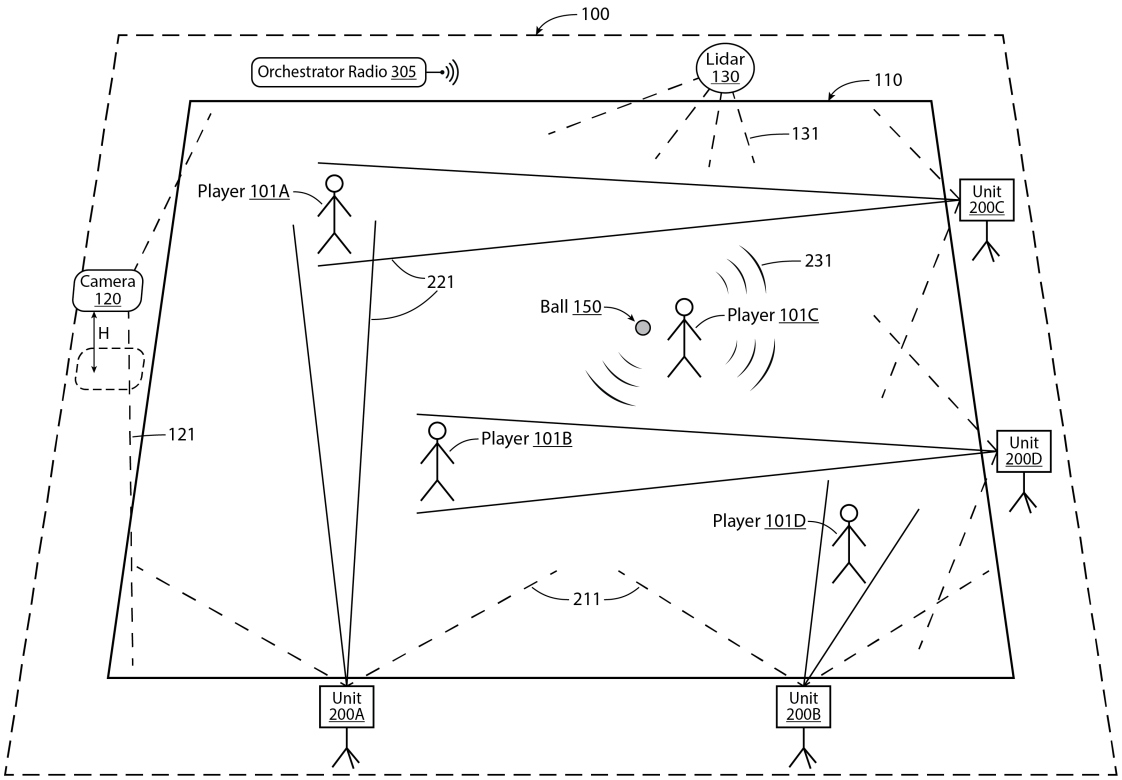


Figure 6.3: LegoTracker sports tracking system overview. Units 200A-D surrounding the game field 110 are working together to reconstruct the position of players 101A-D and ball 150 during the game. LIDAR 130 and elevated camera 120 are used to understand the general context of the game and select active players to track their skeletons.

that are elevated above the field 100 at height H so that their field of view 121 includes the entire game field 110 and players don't occlude each other in the frame as often compared to the tracking units 200A-D located at game field level. The height H for which the principal ray/optical axis of camera 120 will have elevation above the ground field 100 of 20 degrees or more is sufficient. The tracking units 200A-D, in contrast, are located at the level of the game field (roughly half of the player's height above the ground) in order to simplify the design of motorized mirror 209 (only one axis is needed in such configuration). The system is complemented with a LIDAR sensor 130 that is using laser technology for distance measurements

(rays 131) and is placed at the same height as tracking units 200A-D (about half of the player's height) to provide instant estimates of the position of players 101A-D on the game field 110.

As the game progresses, the position of players 101A-D and the ball 150 changes. Each tracking unit 200 can detect in real-time when the object of interest that it is currently tracking is about to go out of the field of view 221 of the zoomed camera 220 (Figure 6.4). This information is used as feedback to motorized mirror 209 placed in front of the zoomed camera 220. Its position is constantly adjusted to keep the object of interest within the field of view 221 of zoomed camera 220. To detect the object of interest in the first place, each tracking unit is equipped with panoramic camera 210 which has a field of view 211 such that it contains any possible field of view 221 of the zoomed camera 220.

By design, each tracking unit 200 is a self-sufficient unit and can decide independently on which object of interest to follow. However, the tracking units are connected using wired (e.g. Ethernet) or wireless (e.g. Wi-Fi) network connection allowing them to share the information about objects that are within their reach and make a more optimal joint decision on which objects should be tracked by which unit. This helps to avoid situations like the one shown in Figure 6.3, where there are four players 101A-D and four tracking units 200A-D but player 101C (with the ball 150 next to it) is not an object of interest for any unit. It is still being tracked by panoramic cameras but is not in the field of view of any zoomed camera that could provide a much higher level of detail for that object. It is possible though, that the decision is made to intentionally ignore this player because he/she is not an interesting (that is, "active") player of the game at a given moment and focus resources on tracking in detail other players (e.g. 101A). Different perspectives of

units 200A and 200C are then used to reconstruct 3D information about the player, such as a 3D skeleton representation of that player.

6.2.1 Unit Design

Figure 6.4 is showing an exemplary embodiment of the tracking unit 200 (all units 200A-D in Figure 6.3 are identical). Its main component is a computing platform 201. We are using NVIDIA Jetson as a computing platform that has a powerful Graphical Processing Unit (GPU) and shared memory between CPU and GPU. Two critical peripheral devices are local storage 202 and network interface 203. Video data 212 and 222 coming from panoramic 210 and zoomed 220 cameras require high bandwidth due to the high-speed nature of sports and, therefore, the high frame rate of the cameras. Video data is compressed by computing platform 201 and stored in relatively slow local storage 202 such as a Hard Disk Drive (HDD). Alternatively, faster local storage, such as a Solid State Drive (SSD) with PCI-Express interface, is used to store video data without compression and with additional metadata (e. g., a precise timestamp or position of the mirror 209 for each frame). This way, computing platform 201 can compress and transfer video data from fast to slow storage on demand. Audio data 233 does not require compression due to its low data rate compared to video data.

Sound is the second most actively involved modality when watching a sports game. Important game events, such as when the ball 150 is hit or caught, are often accompanied by a distinct (loud) sound 231. In the air, sound covers 1 m in about 3 ms, so it is possible to localize the sound source by using a microphone array with a standard sampling frequency (e.g. 48 KHz) and measuring the time delays in sound registration by different microphones. This is a known method

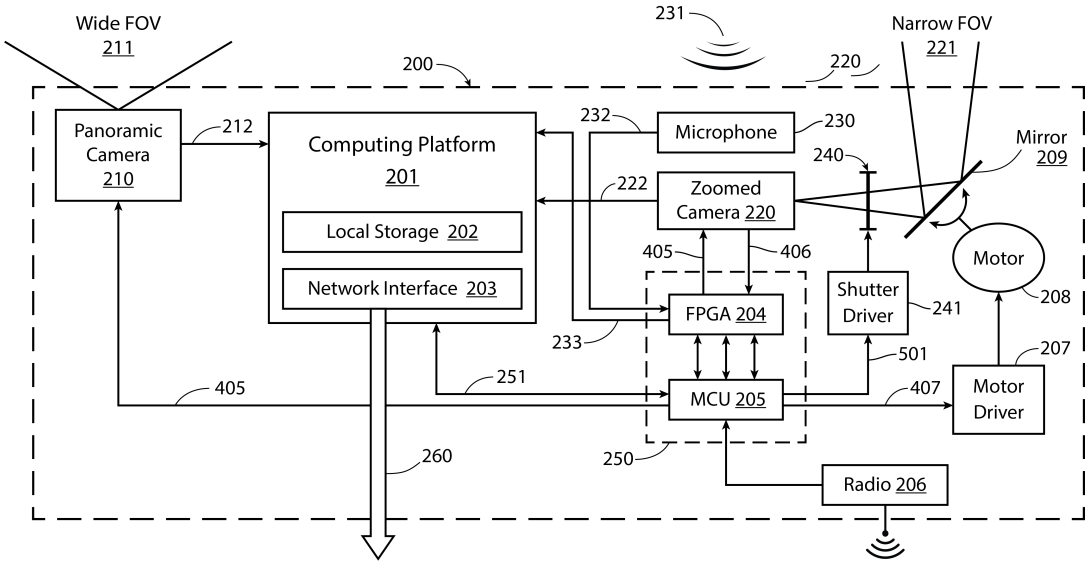


Figure 6.4: Tracking unit 200 combines a panoramic camera 210 for initial identification of players and a zoomed camera 220 with motorized mirror 209 for their tracking in more detail. Microphone(s) 230 are used for the detection and triangulation of loud events. Everything is powered by the NVIDIA Jetson computing platform.

that requires precise synchronization of the microphones and the system includes a wireless synchronization technique for tracking units with an error of less than one period of the sound sampling frequency (see Section 6.2.3).

Each tracking unit 200 is equipped with a microphone 230 and radio module 206. Radio module 206 is passively receiving a timestamp from orchestrator radio module 305 (Figure 6.3) placed somewhere around the game field 110. Because the speed of the radio waves is almost a million times greater than the one of the sound, it is assumed, with negligible errors, that each radio module 206 receives the timestamp simultaneously. This timestamp is then used by microprocessing module 250 to synchronize all the signals sent out to different components of tracking unit 200. In particular, the timestamp received from the radio module 206 is embedded into the original audio data stream 232 and modified audio data 233 is then sent

to computing platform 201. The microphone 230 can be a mono microphone and a timestamp is used to form a second channel of stereo audio data stream 233 that is sent to computing platform 201 via a corresponding interface, such as (but not limited to) a digital I2S interface. Alternatively, the timestamp can be embedded into a stereo audio stream 232 in place of unused least significant bits of each sample, improving in this way the synchronization accuracy by a factor of two without degrading the sound quality. The embedded timestamp is then used to adjust for a slight sampling rate mismatch (due to the manufacturing errors, more details in Section 6.2.3) in audio data captured by different units 200A-D by adding/removing a few audio samples once in a while to match the sampling frequencies as close as possible.

A synchronization signal, such as hardware trigger 405 or camera exposure signal 406, can also be interleaved with the audio data bit stream to further increase the synchronization accuracy between audio and video data streams. It provides superior synchronization accuracy but requires extra processing of the data stream after it was received by computing platform 201. Cameras 210 and 220 have native support of hardware trigger 405 which is synchronized among different tracking units using timestamps received from radio module 206 (Section 6.2.3). When using a wired network interface 203, the wireless synchronization technique of the proposed system can be substituted with Precision Time Protocol (PTP).

Microprocessing module 250 comprises two major components. The first is a microcontroller unit 205 executing a program that is responsible for communication between computing platform 201, the radio module 206, shutter driver 241, and motor driver 207. It is also performing synchronization tasks. The second is a Field Programmable Gate Array (FPGA) 204 responsible for connectivity between the

modules and implements the logic necessary to embed the synchronization signals into the audio data stream 232. FPGA 204 can also be replaced with a Printable Circuit Board (PCB) and standard logic components or a single System on a Chip (SoC), such as Xilinx Zynq, can be used instead too. Connection 251 between microprocessing module 250 and computing platform 201 is established using a standard USB protocol.

Motor driver 207 is a hardware module generating signals that drive motor 208 in response to the motor trigger 407 issued by microcontroller unit 205. The motor trigger 407 is synchronized with camera exposure 406 in such a way that motor 208 rotates the mirror 209 during the time between the consecutive frame exposures (Figure 6.7). Mirror 208 needs to remain still while zoomed camera 220 is exposing the frame to avoid motion blur. In our prototypes (Sections 6.3 and 6.4) motor 208 is a stepper motor and torque transmission is implemented using gears. This reduces the number of oscillations and mirror settling time after the step was made compared to, for example, belt transmission.

6.2.2 Software Architecture

Figure 6.5 illustrates the software architecture of the proposed system. It can be subdivided into parts that are tied to and executed by a particular component of the system (e. g. computing platform 201) and platform-independent blocks, such as inference services 340 and orchestrator 350. The latter can also be executed on computing platform 201 of selected tracking units 200 as well as on a dedicated server. Communication between all blocks is established via network 330 and standard TCP/UDP protocols. Computing platform 201 has significant computing power and network-based architecture enables dynamic load balancing between

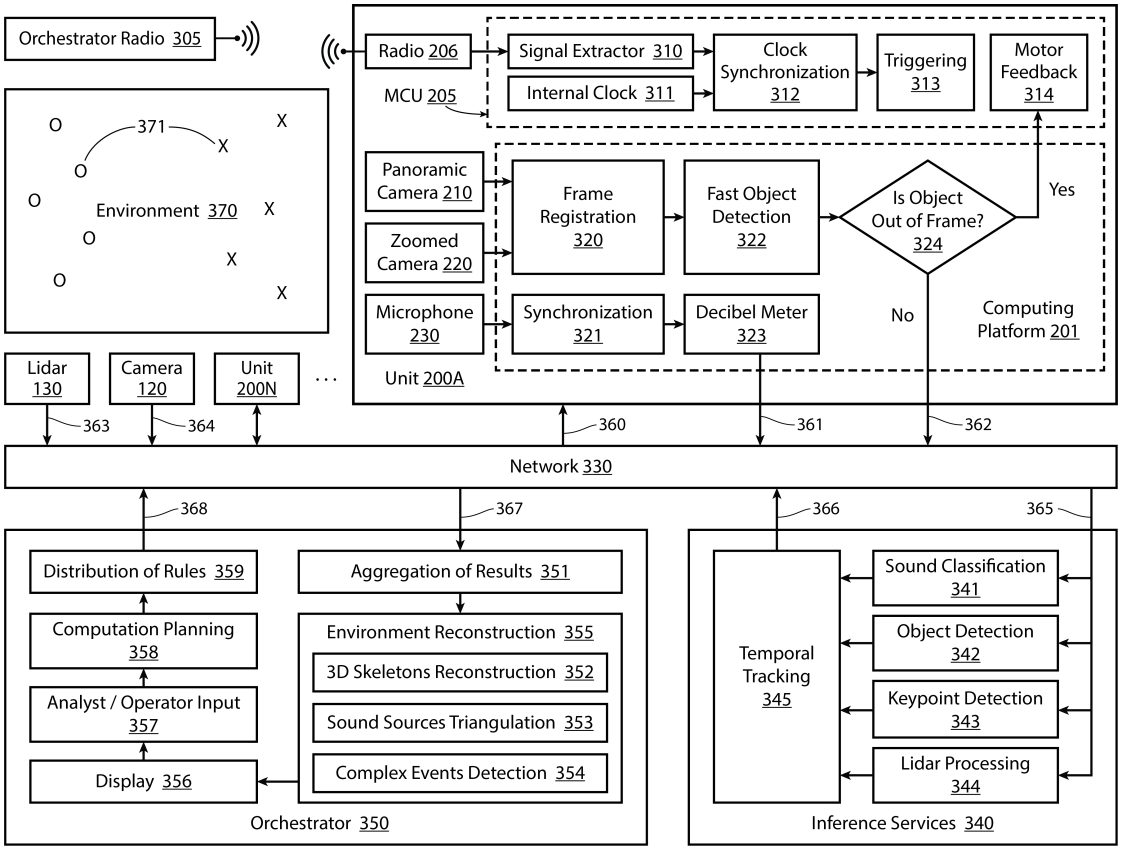


Figure 6.5: LegoTracker software architecture diagram. The software is organized into blocks that implement REIP API so the data acquisition and management can be done by the REIP software framework. Major components, such as inference services 340 or orchestrator 350, are using network interface for easy integration on computing platform 201 of multiple tracking units 200.

the tracking units and eliminates the need for (but does not prevent the use of) a dedicated server. The following is a detailed description of each software component.

The most tied to a particular component is a program executed by microcontroller unit 205. It is driven by internal clock 311 and receives timestamps from radio module 206 distributed by orchestrator radio 305 (Figure 6.6). Internal clock 311 and timestamp decoded by signal extractor 310 are provided to clock synchronization module 312 and the synchronized clock is then subdivided by triggering module 313 to desired frame rate for panoramic 210 and zoomed 220

cameras. The software of embedded microprocessing module 250 is the only part that is not powered by the REIP framework.

When computing platform 201 receives a frame from panoramic 210 and zoomed 220 cameras, they need to be registered with respect to each other. Frame registration 320 is a process of determining the exact position of the frame taken with zoomed camera 220 within the frame from panoramic camera 210. The important property of the proposed design that combines two (zoomed and panoramic) cameras is that the registration error for each frame remains constant over time. The panoramic frame contains some fixed distortions that are calibrated out once and the position of the zoomed frame is refined starting from an estimate based on the position of the mirror 209. The position of mirror 209 is not precisely reproducible but the registration error of zoomed frame is limited by the same value for each frame. Without the panoramic camera, the zoomed frame must be registered with respect to the previous zoomed frame to calculate the absolute position and the error accumulates over time. This is similar to the way the human eye works, which also has a relatively low resolution of peripheral vision (panoramic camera) and sharp central (fovea) vision (zoomed camera).

Once registered, the frames are processed by fast object detection algorithm 322 to determine whether the object of interest is out of frame for the zoomed camera 220. If the answer is positive (Yes) then a message is sent via connection 251 to motor feedback logic 314 residing in microcontroller unit 205 to adjust the position of the mirror 209 so that the object of interest is within a field of view of the zoomed camera 220. If the answer is negative (No) then the frames are sent (362) to inference services 340 for further processing. Frames provided by camera 120 are sent to inference services 340 directly.

A second pipeline executed on computing platform 201 is the one processing sound data. Audio data stream 233 received by computing platform 201 from microphone 230 is processed by synchronization block 321 extracting the synchronization signal and aligning audio samples to a video stream. Synchronized audio is then analyzed by decibel meter 323 to detect any loud events and send results directly to orchestrator 350 for sound source triangulation 353. Audio data is also sent (361) to inference services 340 for sound classification 341. Tracking unit 200 may also contain an array of multiple microphones 230, thus, enabling the localization of sound sources using beamforming techniques.

Inference services 340 is a collection of primarily deep convolutional neural network algorithms for the detection and localization of various objects and sound patterns in data streams. Sound classification 341 executed for the sliding window over the audio stream results in a vector of probabilities for each sound class. The output of object detection 342 is a set of bounding boxes for objects detected in a frame (e.g. a player or a ball) and classified with a confidence level greater than a given threshold. Keypoint detection 343 outputs a list of keypoints and their locations for each detected object including and not limited to a tip of a baseball bat or joints of the player's body such as a knee, elbow, etc. LIDAR processing 344 is performing segmentation of point cloud 363 received from LIDAR sensor 130. Points corresponding to objects of interest provide an independent estimate of their position. The output of each detector/classifier computed for individual frames is then sent to temporal tracking 345 for filtering and formation of temporary consistent trajectories for detected objects.

Orchestrator 350 is a software block responsible for the aggregation of results 351 and high-level reconstruction of environment 370 as well as interaction with the

system operator. Environment reconstruction 355 includes and is not limited to 3D skeletons reconstruction 352, sound sources triangulation 353, and complex events detection 354. 3D skeleton reconstruction 352 can be performed even if the player is tracked by a single tracking unit 200 when standard triangulation techniques are unavailable (see Section 6.3.1). In this scenario, a dedicated neural network is trained on demand for a given viewing direction as the position of the player and tracking unit are known.

Tracking results are then sent to display 356. Display 356 is a display of a dedicated computer executing the orchestrator software block 350. it can also be embedded into tracking unit 200 and the orchestrator software block is then executed on computing platform 201 of that tracking unit. Alternatively, display 356 is a web page that can be displayed on any device (including mobile devices) that is connected to network 330 directly or via the Internet.

There are two ways to affect the operation of the tracking system. A first is direct control of data flow in the system through means of system operator such as enabling/disabling different components of the system, defining which platform is executing inference services 340 for each tracking unit 200, etc. The second is analyst input. The analyst is a person (e.g. team coach or sports commentator) who defines targets, triggers, and other rules of how the system should perform in different game contexts. In baseball, for example, tracking units might be focused on the pitcher before he throws the ball and on outfielders after it was hit by a batter, completely ignoring the pitcher now. Computation planning 358 is a stage when analyst input is taken into account and a set of rules is defined for each tracking unit 200 which are then distributed (359) to the units via network 330. A possible rule for tracking unit 200A is to capture the frame with phase offset

with respect to the other tracking unit 200C. With two tracking units following the same object of interest (e.g., player 101A on Figure 6.3) and phase offset 50 % the effective frame rate at which the object is being recorded doubles. The timing characteristics of the system are described in more detail in the following section.

6.2.3 Wireless Synchronization

As shown in Figure 6.6, radio modules 206A and 206B of two different tracking units 200A and 200B are receiving a timestamp from orchestrator radio 305 practically simultaneously due to the large speed of radio wave propagation in the air (300,000 km/sec). Signal extractor 310 communicates with radio module 206, retrieves the timestamp, and issues a synchronization signal 402. Signal extractor 310 is designed in such a way that the time delay 401 between the moment 401A when radio module 206 received the timestamp and the moment 401B when synchronization signal 402 was issued is fixed and deterministic. Note that the fixed and deterministic time delay 401 helps to reduce the jitter of synchronized clocks 404A-B and, hence, improves accuracy but, in a long run, it is not mandatory for synchronizing the clocks 311A-B that drift with respect to each other due to the slight frequency mismatch (see below).

Each microcontroller unit 205 in tracking unit 200 has an internal clock 311 that is driving the execution of its program. Due to manufacturing errors, there is a difference in frequencies of internal clocks 311A and 311B causing a phase shift 403 between their waveforms 403A and 403B. Phase shift 403 is either increasing or decreasing over time with constant speed depending on how much the frequency of internal clock 311A is greater or smaller than the frequency of internal clock 311B. Clock synchronization 312 is done by aligning the phase of internal clock

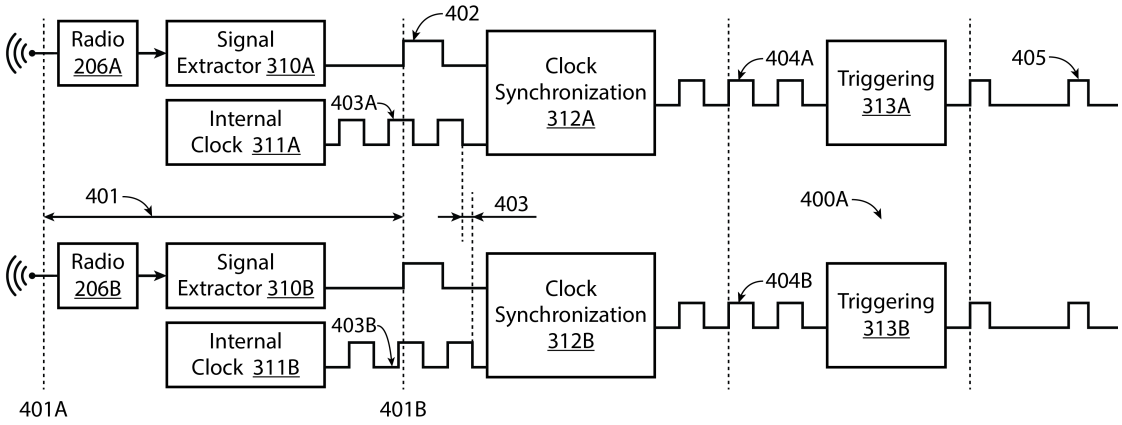


Figure 6.6: Timing diagram of software block executed by microcontroller unit 205 that is responsible for synchronization of tracking units 200.

311 with synchronization signal 402 and implementation of clock synchronization 312 is based on an internal hardware timer of microcontroller unit 205 with the highest CPU interrupt priority. Clocks 404A and 404B are synchronized upon arrival of synchronization signal 402 but keep drifting with respect to each other in between. A high-precision hardware timer can be used with its period adjusted based on the measured time interval between consecutive synchronization signals 402 to reduce the speed of phase drift between clocks 404A and 404B. Finally, the synchronized clock 404 is subdivided by triggering module 313 to frame rate frequency and trigger 405 is sent to the camera.

6.2.4 Double Step Motor Technique

Mirror 209 is placed in front of the zoomed camera 220 and coupled to motor 208 (Figure 6.4). Because of the finite exposure used by the zoomed camera 220 (i.e. less than 5 ms at 200 fps), any movement of mirror 209 during the frame exposure will cause it to be blurred horizontally (Figure 6.7). Therefore, any change of field of view 221 of zoomed camera 220 needs to be done in between the frame exposures

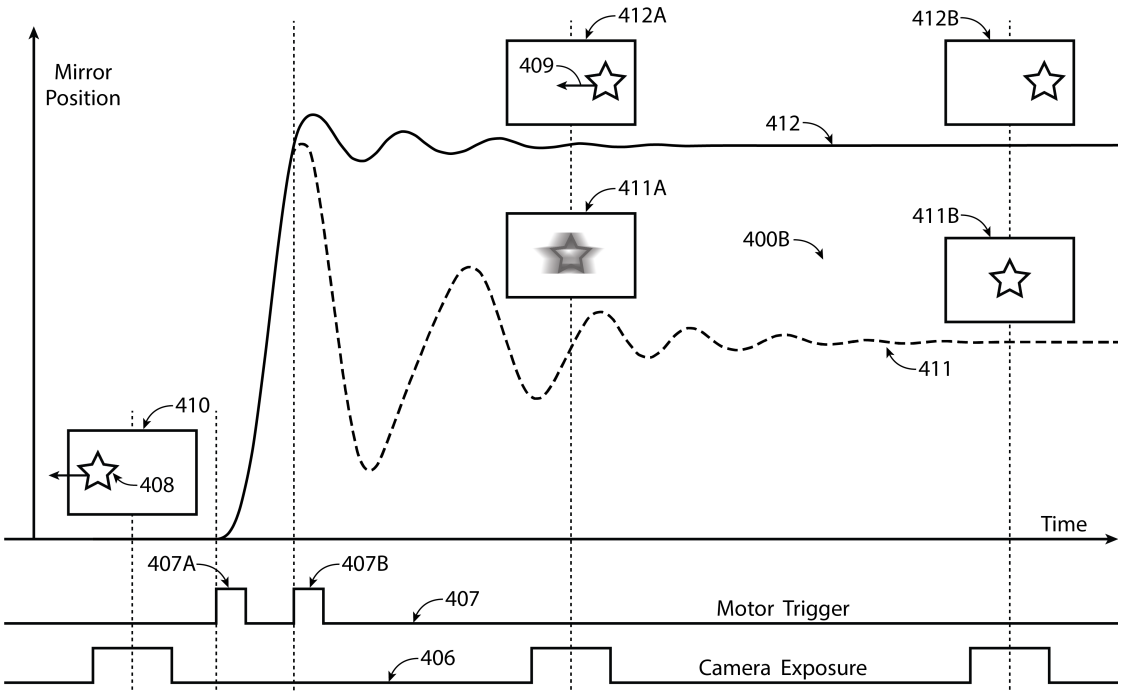


Figure 6.7: A transition curve for single (411) and double (412) steps performed by a motor 208 with a waveform of motor trigger 407 and camera exposure 406 signals below. The double-step technique enables faster adjustment of the camera view while minimizing the motion blur. The span of the time axis is about 10 ms.

and mirror 209 must be still during the actual exposure. We present a technique for increasing the frame rate of zoomed camera 220 while increasing the speed at which its field of view 221 is being adjusted at the same time.

Frame 410 is the last frame taken by zoomed camera 220 before the object of interest 408 moving in direction 409 is about to get out of its field of view 221 (Figure 6.7). Shortly after, microcontroller unit 205 receives feedback from computing platform 201 and issues a motor trigger 407A. Motor driver 207 reacts to the motor trigger 407A and configures a new position 411. Because forces pulling mirror 209 to the new position 411 are symmetric with respect to that position, mirror 209 is oscillating a few times around the new position 411 before its kinetic energy is dissipated due to friction and it stops. This causes the following frame

411A to be blurred significantly, reducing the effective frame rate by a factor of two.

Alternatively, the second motor trigger 407B can be issued when mirror 209 is at an amplitude position close to position 412 of the second step. Motor driver 207 then quickly configures position 412 and mirror oscillations attenuate faster because of the smaller initial amplitude. The consecutive frame 412A is not blurred and the field of view 221 of zoomed camera 220 is adjusted by double the amount of a single step. Such a technique can be extended for triple and multiple steps as well as multiple axes, enabling optical tracking of very fast (100 mph+) moving objects such as a baseball ball during a pitch.

6.3 First LegoTracker Prototype

Figure 6.8 is showing our first limited prototype of the tracking unit 200. For this iteration, we decided to limit ourselves to a static variant of a high-speed camera (without a motorized mirror) and focus on the synchronization feature. The prototype includes a high-speed camera comprising of a Basler Ace acA1300-200um color sensor with USB 3.0 interface and Computar 12.5-75mm varifocal C-Mount lens. It also features a momo MEMS microphone with an I2S interface. We synchronize the two using a version of techniques described in Section 6.2.1 for mono audio with custom PCB for the embedding of the camera trigger/exposure signals into the second channel of the I2S microphone pair. We are using the I2S0 audio interface available on the expansion header of NVIDIA Jetson TX2 which serves as our Linux-based computing platform. The data is saved onto a 1 TB HDD drive with a SATA interface and a regular USB microphone is used as a baseline for audio synchronization.

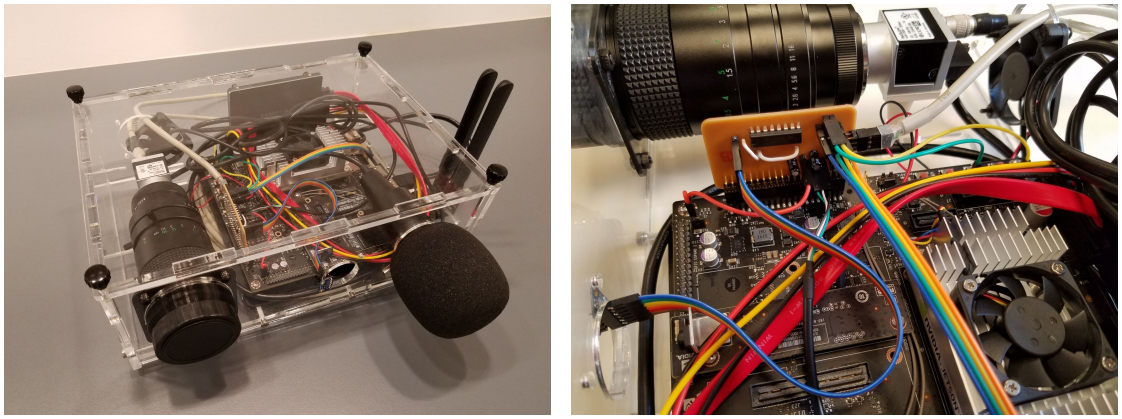


Figure 6.8: First LegoTracker prototype featuring a high-speed camera and a synchronized mono audio stream. A pair of such units was tested during the training session of the Texas Rangers baseball team.

We have built two instances of this prototype and tested them during the May training session of the Texas Rangers baseball team in Frisco, 2018. The objective was to measure optimal camera parameters for recording such fast sporting events as a baseball pitches (the speed of the ball reaches over 100 mph for professional players). The units were spaced 50 feet apart at typical recording locations and synchronized using dedicated wires and the camera's external trigger functionality. Basys 3 FPGA board was used to generate the common trigger for recording at 240 frames per second and 1024x768 pixels image resolution.

It was discovered that with our choice of optically strong lenses and due to the favorable weather conditions during a typical baseball game, frame exposures of less than 1 ms can be used for a high-speed camera, which minimizes the motion blur and is favorable for view adjustment with motorized mirrors as proposed in Section 6.2. We also learned that the sound volume level of the ball caught by a catcher with a glove is also sufficient for the detection of a pitching event even if the batter missed. More analysis of the pitcher's movement during the pitch is presented in the following section.

6.3.1 Pitcher's Pose Reconstruction

Pitcher's pose reconstruction can be broken down into several steps:

- Initial object detection of human class to localize the pitcher;
- Cropping a padded region around the detected pitcher's bounding box;
- 2D keypoint detection, such as feet or knees;
- Elevation of a 2D pose into a 3D skeleton using a pre-trained spring model.

There are multiple object detection models ranging anywhere on the accuracy vs inference time trade-off scale. Since we are aiming to build a network of independent tracing units that are doing real-time object detection and tracking, our choice of object detection model naturally fell onto Single Shot Multi-Box Detector [86]. MobileNet SSD v2 [87] is providing a great balance between the performance and accuracy for embedded systems. We are using an open-source repository jetson-inference[1] by NVIDIA that includes an implementation of MobileNet SSD v2 optimized for Jetson platform with TensorRT.

Prior to keypoint detection, one needs to crop the image around the detected bounding box of the pitcher. Our experience shows that small perturbations of the input image (either from frame to frame or due to the cropping factor) can have a dramatic influence on the accuracy of keypoint detection. We are using OpenPose [85] as our pose/keypoint detector. It is also worthwhile to skip the final thresholding step until after tracking of local maximum for keypoint detection on the predicted heat maps. Such an approach produces more temporally consistent results.

[1] https://github.com/dusty-nv/jetson-inference

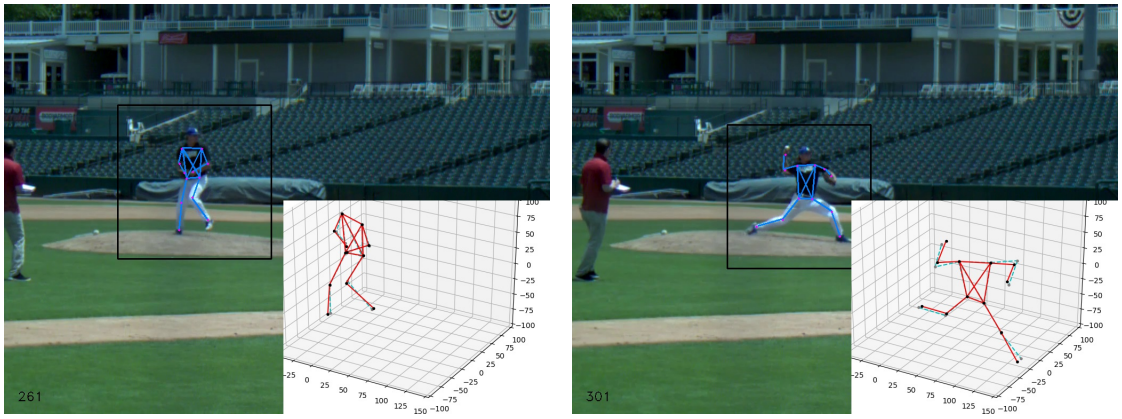


Figure 6.9: 3D pitcher's pose is being reconstructed from 2D joints of a single camera view. Existing mocap data is used as a prior for spring model optimization. The blue skeleton is the model's prediction and the red one is shown after optimization.

Finally, the detected 2D pose of the pitcher (Figure 6.9) can be elevated into 3D even if only a single camera view is available. To do so, we train for a given camera view a small neural network that is predicting the depth value for each joint with respect to the root (middle of the heaps) from that viewing direction. The network is trained based on existing mocap [88] data of the baseball pitch. Because the pitch is an anisotropic motion with a specific pitching direction, the network needs to be retrained for every camera view/position which is static but unknown prior to unit placement on the game field.

We augmented the dataset using a spring model of the human skeleton. In this model, bones are modeled as rods with springs connecting the joints along those rods/bones. In the rest pose, the length of each spring is equal to the average length of corresponding bones in the human skeleton. The sum of the potential energy of bone springs in the augmented pose is used as an optimization loss to generate more valid human poses similar to those observed during the pitch. We also apply this optimization as a filtering step after the depth of joints was predicted by the trained model. Additionally, we modify the pose optimization

during post-processing by including a pose from the previous frame to simulate the effect of inertia and achieve temporally smoother results. Sample results are shown on the bottom-right of frames 261 and 301 in Figure 6.9.

6.4 Second LegoTracker Prototype

Our second LegoTracker prototype (Figure 6.10) is reusing the same high-speed camera and computing platform as the first prototype. To achieve fast adjustment of the field of view for a high-speed zoomed camera, one needs to minimize the inertia of any moving parts involved. Because the camera sensor with lens assembly is one of the heaviest parts of the tracking unit and weighing about 1 kg, we place a motorized 2x3" mirror in front of the zoomed high-speed camera for dynamic adjustment of its viewing direction. We have chosen a Nema 23 stepper motor with 1.2 $N \cdot m$ of torque and a gear ratio of 1:2 to power the mirror. The motor driver used is TB6600 with an LM2596-based 4-Channel switching power supply module that is also used to power the microprocessing module as well as cooling fans.

This time, we have stereo audio with a pair of Adafruit I2S MEMS microphones. However, audio muxing is now performed by a Cmod-A7 FPGA breakout board from Digilent. We opt for an FPGA solution because we are using the second version of the synchronization technique (Section 6.2.3) where camera trigger and exposure signals are embedded as the least significant bits into the audio data stream. Because the I2S0 interface of NVIDIA Jetson TX2 supports 32-bit audio, we are not losing on audio quality as the microphones are only utilizing 24 bits per audio sample.

The prototype includes a full stack implementation of the multiprocessing

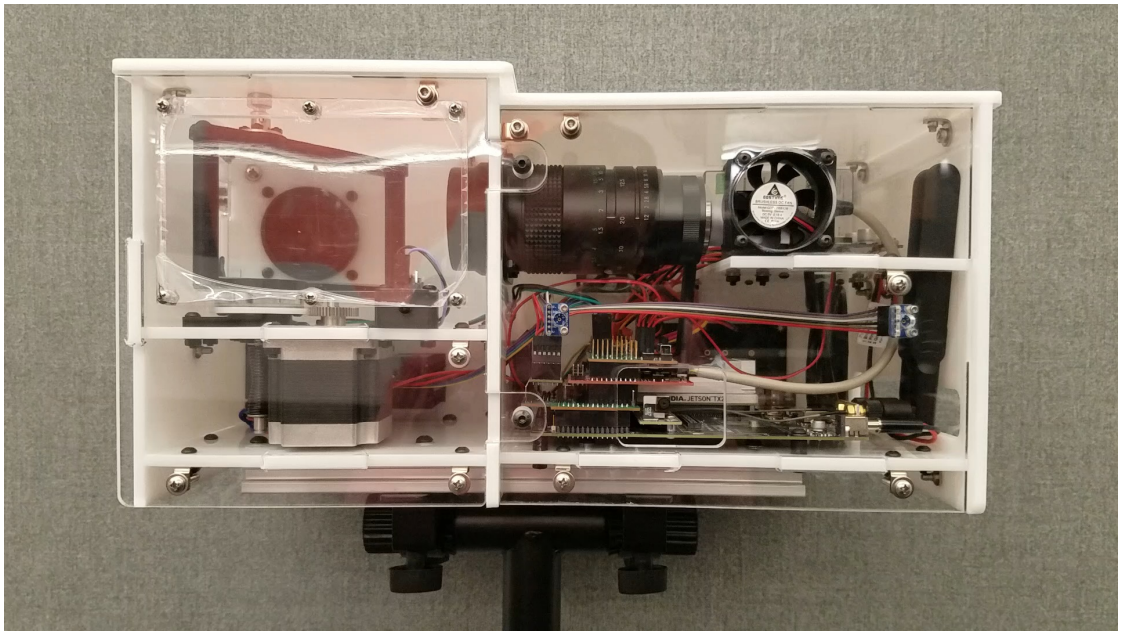


Figure 6.10: Second LegoTracker prototype that is reusing the same hardware component with the addition of a motorized mirror in front of the high-speed camera. It also includes a full stack implementation of a microprocessing module including a microcontroller, an FPGA, and custom PCBs.

Figure 6.11: Portable orchestrator radio based on nRF24L01P+ 2.4 GHz radio module with touch screen booster pack of Tiva C microcontroller as the input device.

module with nRF24L01P 100mW 2.4GHz radio module for wireless synchronization. All the components (including FPGA) are connected together with help of custom PCBs to ARM Cortex M4 based microcontroller Tiva C. SPI protocol is used for communication with the radio module and UART bridge via an FPGA for debugging access to the Serial Console of the Jetson computing platform. Another custom PCB is used to route the trigger/exposure signals to/from the high-speed camera as well as control signals for the motor driver. The built-in 5 MP @ 30 FPS camera of Jetson TX2 is being used as a wide-angle camera in this configuration.

Figure 6.11 is showing as orchestrator radio 305. It is based on the same Tiva C microcontroller and nRF24L01P+ radio module. It also features a BOOSTXL-K350QVG-S1 320x240 touchscreen booster pack by Texas Instruments as an input device. The controls include Synchronization and Recording flags as well as the selection of one out of two prototype instances for manual control of their corresponding mirrors during system setup.

Just like every custom mechanical hardware component, the motorized mirror requires calibration for optimal adjustment of viewing directed of the high-speed camera with minimal motion blur. Figure 6.12 is showing a response function of the motorized mirror for a single and double-stepping technique described in Section 6.2.4. The optimal delay for the second step pulse is $2.5ms$ in this case and there is obviously not enough friction to quickly attenuate the residual oscillations of the mirror. Further analysis reveals that the acrylic enclosure of the prototype does not provide sufficient stiffness which contributes to the residual oscillations.

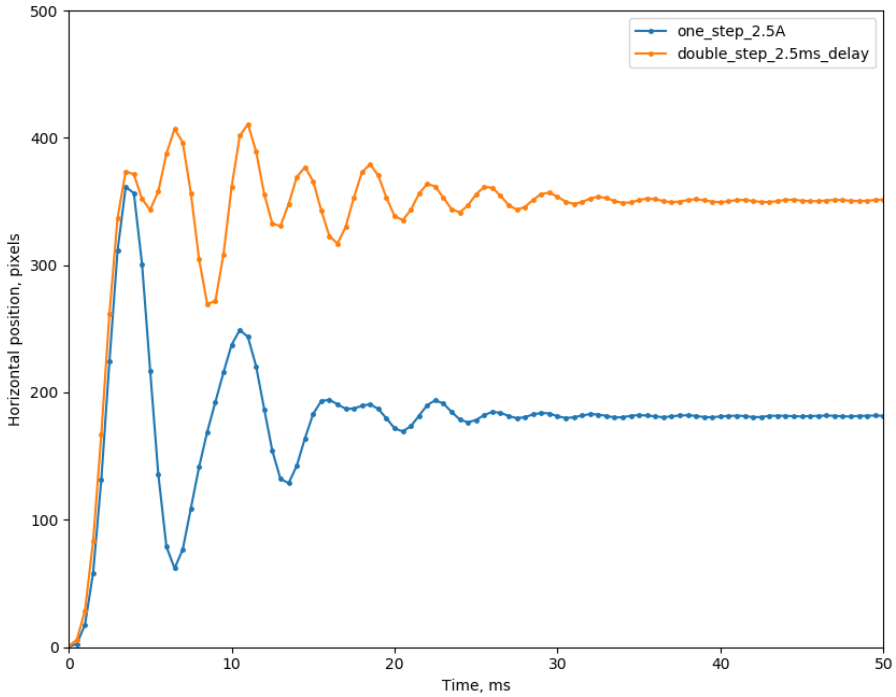


Figure 6.12: Mirror response function. Viewing direction can be adjusted at double speed (orange curve) with the double-stepping technique but requires accurately calibrated timing of the second step pulse.

6.4.1 Data Samples

Figures 6.13 and 6.14 are showing the video and audio data samples acquired using the second prototype in the city park setting. A number of issues have been identified during the testing of this prototype as it was our first attempt at implementing all of the features proposed in Section 6.2.

With regards to the cameras, the most prominent issue is the lack of dynamic range when recording with a high-speed camera, partly due to the reduced bit resolution of the sensor when recording at very high frame rates and also because of the shadows (Figure 6.13, right). These issues were not discovered while testing the first iteration of the prototypes because of very favorable weather conditions during the Texas Rangers training session. Secondly, in the complete field-of-view-

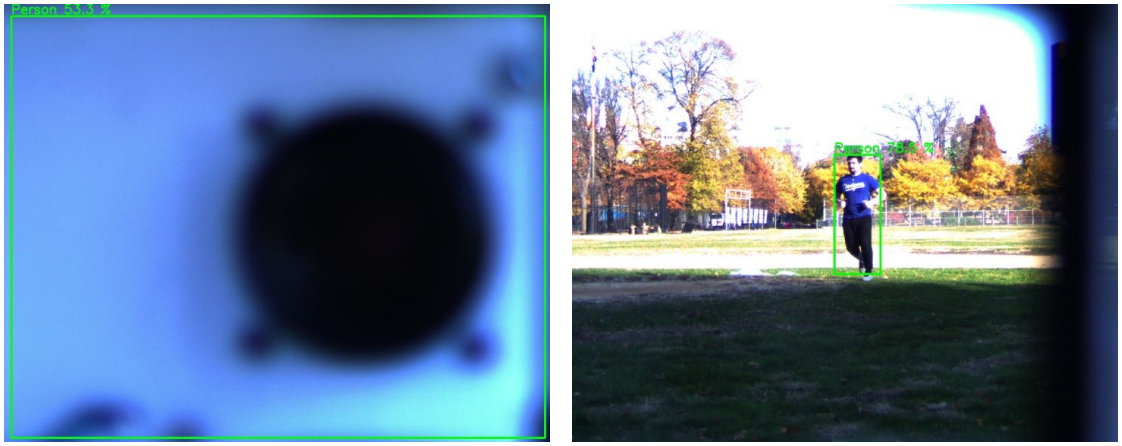


Figure 6.13: Video samples with object detection overlay acquired using the second prototype. Of notice is the camera's self-identification as a person and poor dynamic range in presence of sun shadows, which negatively impact the real-time tracking.

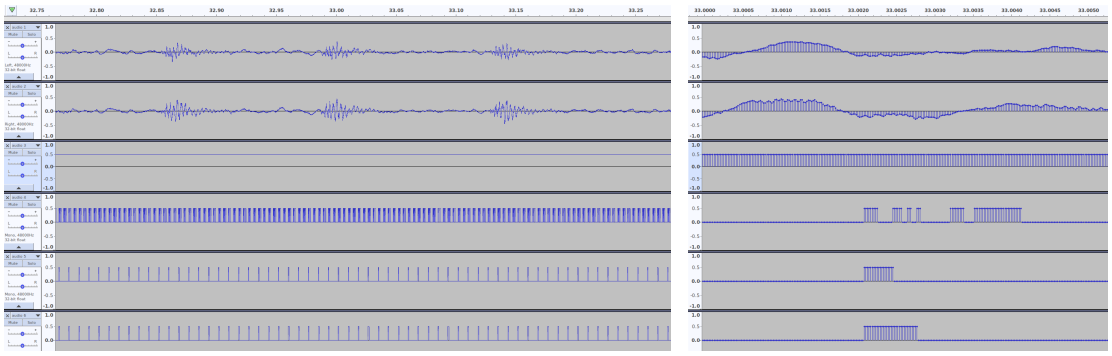


Figure 6.14: Audio samples from stereo microphone used in the second prototype with zoomed view on the right. Extra information contained in the least significant bits of the data has been extracted into separate channels (top to down): stereo audio, data valid flag, serialized timestamp, camera trigger, and exposure signals.

adjustable design zoomed camera has a chance of recording its reflection during the homing procedure of the mirror (recovery of the mirror's position at the start of the recording using limit switches) and locking in that position since, apparently, the tracking unit self-identifies itself as a person (Figure 6.13, left), although with low confidence. Finally, the non-optimal performance of both zoomed and built-in wide-view cameras also results in low confidence of the object of interest detection

and poor tracking. The neural network inference performance on a rather dated Jetson TX2 computing platform introduces substantial latency into the mirror feedback loop which, coupled with low confidence / intermittent detections, often leads to loss of the object of interest from the zoomed camera's field of view.

The audio recording implementation performed well with not only the synchronization signal being embedded into the audio stream as least significant bits but also auxiliary camera signals, such as trigger and exposure. However, the mechanical coupling between the microphones and mirror assembly has introduced a significant amount of background noise when the zoomed camera's field of view is being actively adjusted (Figure 6.14). This makes it extremely difficult to decipher the real audio signal containing the ball hit or catch sound from the raw audio stream. An elastic suspension would need to be introduced in the subsequent prototypes to mechanically decouple the microphones from the main frame of the tracking unit.

6.5 Third LegoTracker Prototype

Equipped with all the knowledge and experience from findings during the testing of the first two tracking unit prototypes, we have designed our third and final version that addresses all of the previously discovered issues (Figure 6.15).

It features Teledyne FLIR ORX-10GS-32S4C 12 bit sensor with Computar E5Z2518C lens as high-speed zoom camera and FSCAM_CU135 from e-con Systems is serving as a 4K wide-angle camera. The choice of the 12 bit sensor is driven by the need for a higher dynamic range in images and the camera has a powerful ISP (Image Signal Processor) to do live gamma and exposure correction. Pre-processed

Figure 6.15: Ray traced rendering of the third LegoTracker prototype CAD model. See photo of a built unit during field testing in Figure 6.18.

images are then transmitted via 10 Gbps Ethernet connection in 8-bit Bayer format at a rate of 150 fps and 1920 x 1200 resolution. Computar lenses with up to 135 mm variable focal length and 3.45 μm pixel size of the sensor provide maximum angular resolution such that a player 100 meters away can be recorded at 700 pixels vertical resolution at max zoom setting, double that of the second prototype and tenfold that of the wide-angle camera. The wide-angle camera also has an onboard image processor providing high dynamic range images and was optimized for relatively high angular resolution (4K across 60 degrees horizontal field of view) as well. This way, far players can be reliably detected as objects of interest for tracking.

We are using NVIDIA Jetson Orin as the computing platform for the third LegoTracker prototype. It offers more computing power which is important for minimization of latencies during live tracking. We have also opted for MCHStreamer as our audio interface for support of more channels and ease of implementation.

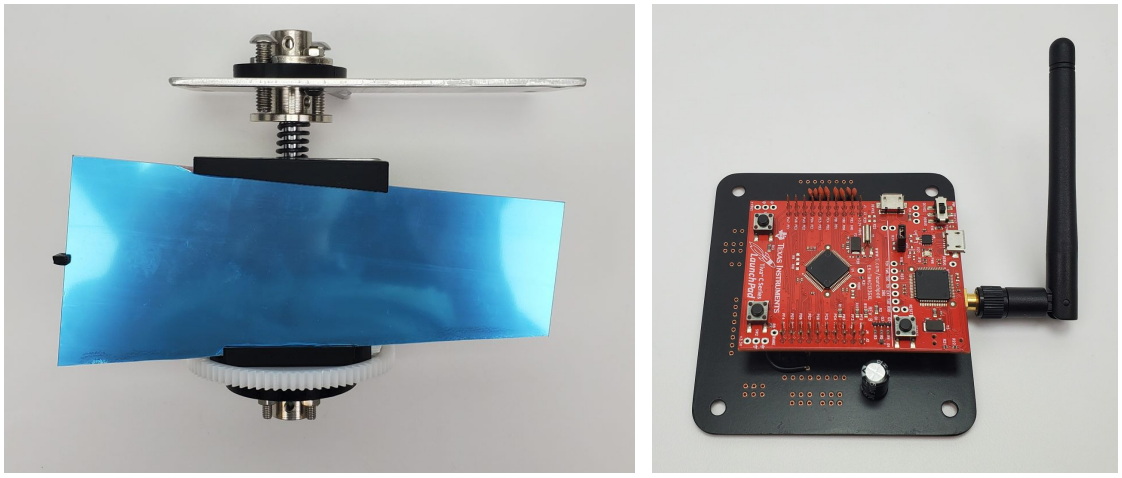


Figure 6.16: Upgraded mirror assembly (left) and synchronization module (right).

There is no more need in using FPGA for muxing of the audio and synchronization signals (i.e as least significant bits), they can be sampled directly as independent channels (MCHStreamer supports up to 16 audio channels). More on the customized hardware components of this prototype is found in the following section.

6.5.1 Construction

Figure 6.16 is showing an upgraded mirror assembly (left) as well as a synchronization module (right) used in the third LegoTracker prototype. We are using the same radio module and Tiva C microcontroller for synchronization back compatibility with the previously developed master radio. The PCB design has been greatly optimized though, so we only need one custom PCB and it is resilient to electromagnetic interference (EMI) generated by the motor driver. We included debouncing capacitors and digital transistors were necessary to combat the EMI.

The mirror assembly (Figure 6.16, left) is the most optimized component in the latest prototype. It includes two major improvements. The first is the rotational

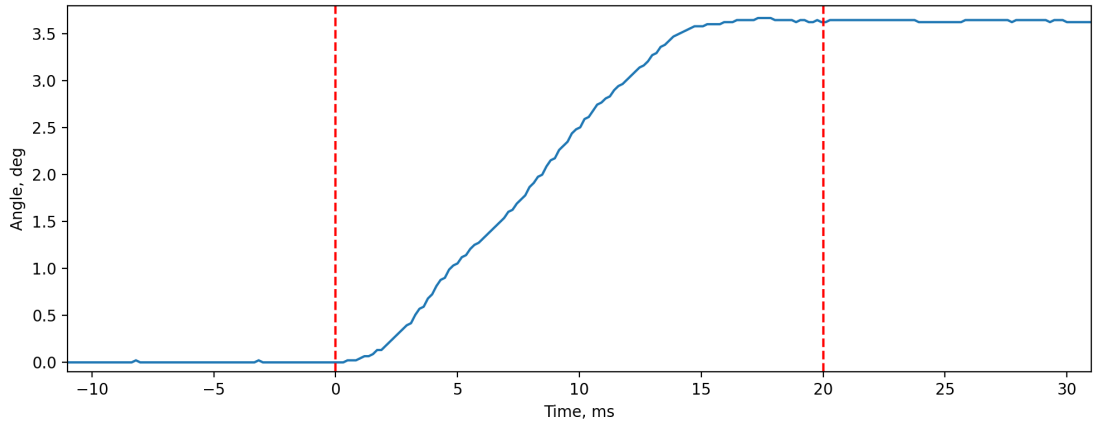


Figure 6.17: Improved mirror response. The revised design of the mirror assembly provides sufficient friction torque and rotating axis stability for oscillation-free view adjustment using the double-step technique (in less than 20 ms).

axis which is no longer a part / defined by the assembly itself but is rather an integral part of the ruggedized main aluminum frame of the unit. Combined with quality bearing it eliminated any residual oscillations in the vertical direction after the camera's field of view has been adjusted.

The second major improvement is the adjustable constant friction torque thanks to the spring-loaded axial force generator with washer bearings. We have also switched to using larger gears of module 1 (metric). Combined with a heavy-duty spring, it provides us with a wide range of friction torques for calibration of the mirror transition time. As shown in Figure 6.17, we have achieved an oscillation-free field of view adjustment in less than 20 ms when using a double-step technique – almost double the speed of the second prototype.

These improvements make it possible to record at a rate of up to 50 frames per second without any motion blur while adjusting the zoomed camera's field of view at the same time! After accounting for the gear ratio and mirror effect, the field of view adjustment speed in this mode is 45 degrees per 0.4 seconds (pitch fly time).

Finally, the motor driver has been upgraded to a model supporting half-current idle mode (Stepperoline DM542T) to reduce the power consumption of the tracking units. The mirror driving motor has the same specification as in the second prototype (Nema 23 with 1.2 $N \cdot m$ torque) and the microphones are now attached to the front cover using rubber suspension to minimize the parasitic vibrations coming from the motor and recorded as sound.

We have also ensured that internally the unit is black to prevent any sun glares or parasitic light and the outside is white to prevent overheating during operation under direct sunlight. Three 80 mm fans and auxiliary heat sinks help keep cameras and other components under normal operating temperatures.

6.5.2 Field Testing

We have manufactured a total of three instances of the latest prototype version which have been field tested in the Commodore Barry Park together with the LIDAR sensor (Figure 6.18). Two tracking units were placed next to the second and fourth bases and the third unit as well as the LIDAR sensor were located next to the home plate. Two people were performing sports actions such as throwing a baseball pitch around the third base area.

The objectives of the testing were to evaluate the reliability of the tracking system (motor feedback loop) so that the units (and zoomed cameras in particular) are always recording data that contains a signal of interest (i.e. the person that possesses the ball). We did also perform quantitative measurements of the players' performance using both zoomed and wide-angle cameras for comparison to see the benefit of the proposed system design. Lastly, we were seeking to prove the feasibility of player tracking using very sparse 3D point clouds generated by the LIDAR sensor.

Figure 6.18: Third LegoTracker prototype with improved performance (Full HD recording at 150 fps). The photos were taken during in-field testing.

6.5.3 Projected Speed Measurements

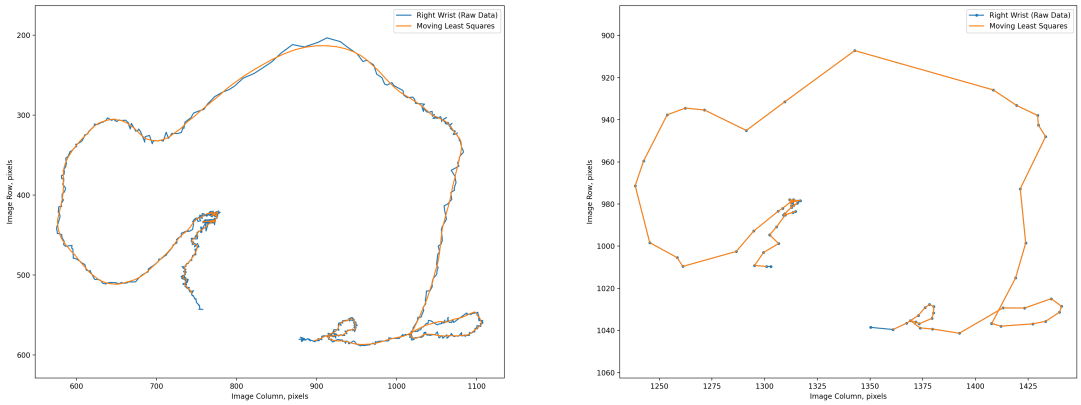
As a quantitative measure, we have chosen to use the projected speed, which is the speed the players were developing during the pitch in a plane perpendicular to the viewing direction of the camera. Such a choice is motivated by the ease of calibration and the ability to evaluate the tracking units independently. Figures 6.19 and 6.20 are showing two examples of such pitches by each student player.

We are using the right wrist trajectory (both players were right-handed) as a proxy for ball speed measurements which we assume to be proportional to the wrist speed. Pose estimation necessary for wrist trajectory reconstruction was performed in post-processing using HRNet [76] as it is much more computationally expensive compared to the object detection required for real-time tracking.

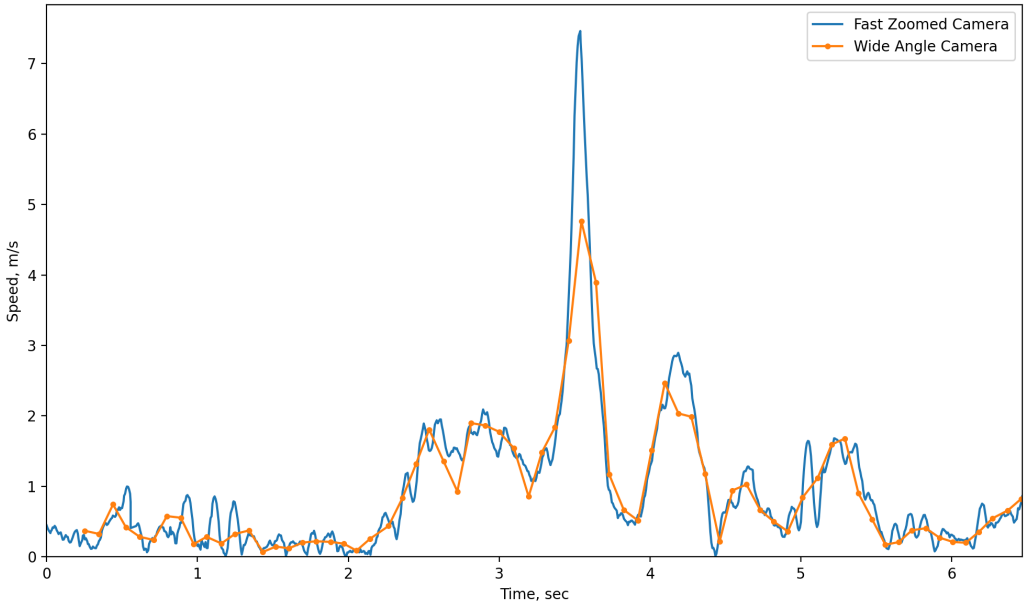
Trajectories that are shown in Figures 6.19b and 6.20b have been frame-by-frame validated to contain correct pose estimation. Combined with a high frame rate of the zoomed camera and low noise in wrist detection due to the high resolution, we treat the trajectory estimated from the footage of zoomed camera as a ground truth.

We further refine the estimated right wrist trajectories using the moving least

(**a**) Video Frames



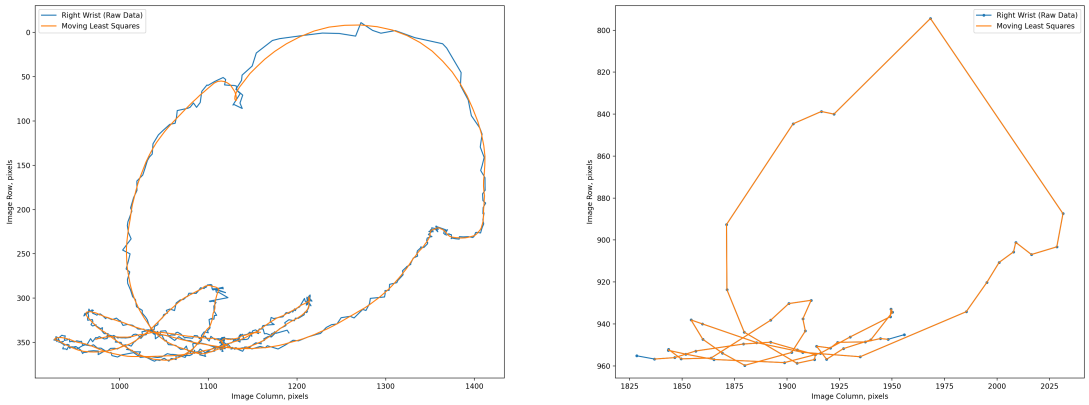
(**b**) Wrist Trajectories



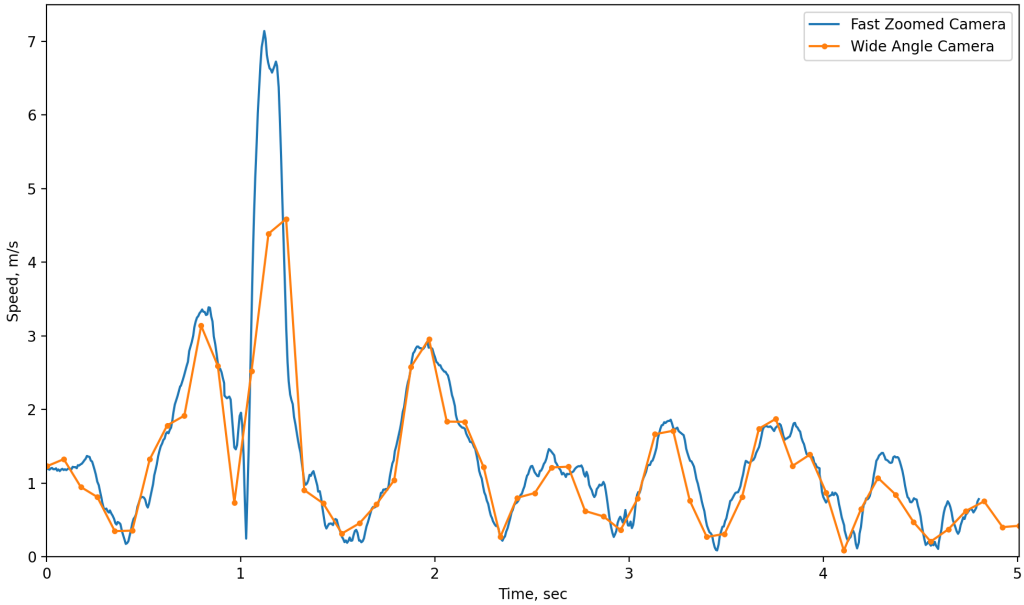
(**c**) Projected Speed

Figure 6.19: Pitch example 1 as recorded by zoomed (left) and wide (right) cameras.

(**a**) Video Frames



(**b**) Wrist Trajectories



(**c**) Projected Speed

Figure 6.20: Pitch example 2 as recorded by zoomed (left) and wide (right) cameras.

squares method [89]. For each frame, we are using 0.1 seconds of footage before and after the frame to fit a polynomial model. Parameters of the best fit are then converted into a reduced-noise trajectory and a projected speed measurement shown in Figures 6.19c and 6.20c (by plugging in t=0 into the fitted polynomial expansion of the local trajectory and its first derivative correspondingly). Because of the limited frame rate of the wide-angle camera (≈ 11 fps on average) we are using a second-degree polynomial model in this case which is equivalent to the constant acceleration physical model. For the zoomed camera, we are using a higher-order polynomial to fully capitalize on its rich temporal resolution.

Although projected speed measurements based on both zoomed and wide-view camera footage generally agree there is a significant, and consistent, underestimation of the peak speed developed during the pitch when measured using a wide-angle camera. This is due to the lack of temporal resolution necessary for quantitative analysis of such high-speed phenomena as a baseball pitch.

Another example of a phenomenon that is not just impossible to quantify correctly but to detect in the first place is dribbling of the ball. An example can be seen in Figure 6.19c around the first second mark performed by the first student. Both temporal and spatial resolution limitations are contributing to this scenario as the angular resolution of the wide-angle camera was three times lower than that of a zoomed camera during the field testing (up to 10x possible depending on the configuration). This highlights the benefits of the proposed system design but which depends on the real-time analysis of the captured data and that was achieved using REIP SDK.

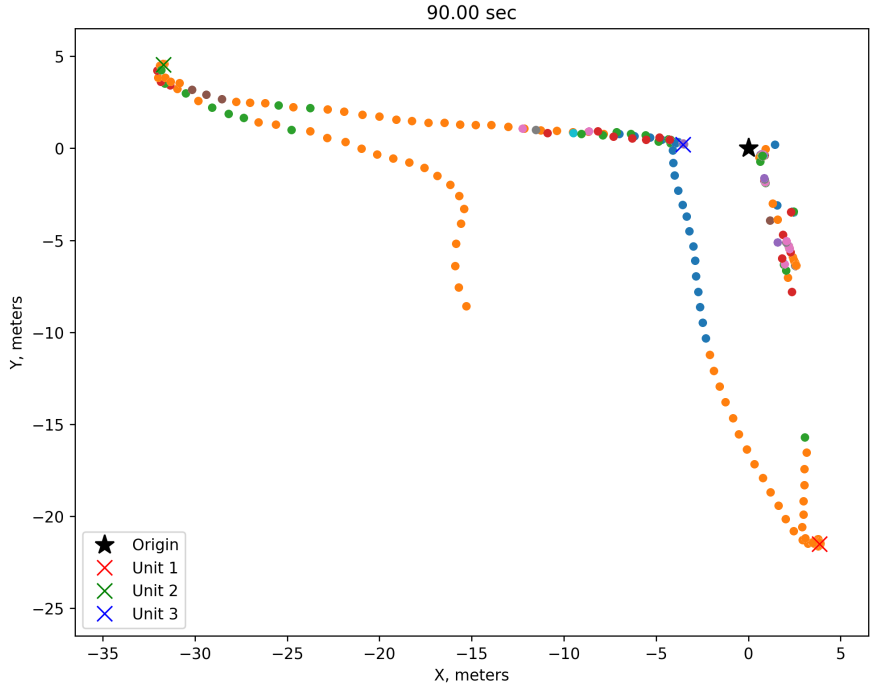


Figure 6.21: Walking trajectory during calibration as recorded by LIDAR sensor.

6.5.4 LIDAR Based Tracking

LIDAR sensor provides the unique benefit of directly providing 3D point clouds of the environment at a reasonable frame rate and which, unlike video data, are much easier to process in real-time with low latency. We are using an Ouster OS1-16 LIDAR sensor capable of generating a 16 by 1024 full circle point cloud at a 20 Hz refresh rate. Our sensor has a tight beam configuration with a vertical angular resolution of 1 degree and a horizontal resolution of 0.35 degrees.

LIDAR data acquisition and processing is powered by REIP SDK and executed on Linux based orchestrator machine/laptop. The acquisition pipeline is linear and contains the following blocks: (1) Sensor block receiving the data packets transmitted by the sensor via UDP protocol, (2) Parser block that is organizing chunk data from individual network packets into complete scans, (3) Formatter block

that is applying sensor calibration and converting raw data into a ready-to-use 3D point cloud, (4) Background Subtraction block eliminating any points that belong to background, (5) Clustering block grouping individual distance measurements into tracked objects and, finally, (6) Plotter block visualizing the tracked object with sufficient amount of measurements. An example of the first 90 seconds of the person's walking trajectory during calibration of the tracking units (using an 800x600 mm Charuco calibration board) is shown in Figure 6.21.

Background Subtraction block works as follows. A short scan is acquired without background subtraction and stored using the default NumpyWriter block from the REIP SDK utility blocks library. The NumpyReader block is then used to replay the scan, select an interval where no one is present in the scene and pass the data to the Background Computation block. The outputs of the Background Computation block are statistical mean, standard deviation, etc. measurements for static background in each laser direction (if detected). This data is then used as a configuration for the Background Subtraction block during the real data acquisition.

For object detection, we are using a modified union-find algorithm that accounts for additional depth parameters as measured by the LIDAR sensor. When a cluster is detected in the point cloud with a total number of adjacent points greater than a predetermined threshold (typically greater than 2), an object is defined that is located in the middle of said cluster. Several additional objects are visualized in Figure 6.21 to the right of the LIDAR origin. These were the remaining member of the team hiding behind the protective net which results in an unwanted segmentation.

Overall, we conclude that the LIDAR sensor is a useful, and functional, addition to the tracking system where ambiguity needs to be resolved as to which objects of interest need to be tracked by individual tracking units.

6.6 Discussion

The differences between the movements of professional players are subtle and richer player tracking can reveal details of player behavior and game patterns that might change the way coaches manage players or plan strategy. We presented a tracking system that can track a large physical area at high resolution (that is, being able to resolve detailed movement of people and objects, such as that required to compute skeletons), in contrast to existing tracking systems that are typically limited by a single point representation for each player when applied at a large game field.

Simply putting a camera overviewing the entire field, for example, limits the achieved spatial resolution to the size of the field divided by the resolution of the camera. A camera focused (that is, "zoomed") on the object of interest would not have this problem but in this case, the tracked object might leave the field of view of the zoomed camera. The observation was that players are sparsely distributed across the game field, which is also flat. We exploited this observation to maximize the signal-to-noise ratio in our system, which we define as the fraction of pixels corresponding to a player over the total resolution of the recorded image. The s/n ratio was 5.8% for zoomed camera in pitch Example 1 (Figure 6.19) and 7.8% in pitch Example 2 (Figure 6.20). In contrast, the same s/n ratio was 0.23% and 0.33% correspondingly for the wide-angle camera. That is 24 times more efficient utilization of the data bandwidth which translates into times higher spatial resolution for the zoomed camera as well as an order of magnitude higher frame rate, both of which are essential for correct estimation of the projected speed.

Another observation is that modern embedded systems have accumulated a substantial amount of computational power. We, therefore, designed our system

in a modular fashion so that the data processing, including centralized analysis of aggregated data, can be distributed across the actual tracking units acquiring that data. This eliminates the need for a dedicated server or huge amounts of cloud resources. Moreover, powered by the REIP SDK data acquisition pipeline can also include a real-time analysis, which we also capitalize on. Without a real-time feedback loop for the motorized mirror, it would be impossible to keep the object of interest within the field of view of the zoomed camera and achieve the aforementioned efficiency in data bandwidth utilization, which directly translates into the amount of tracking details and useful inferences that can be therefore computed.

6.6.1 Limitations

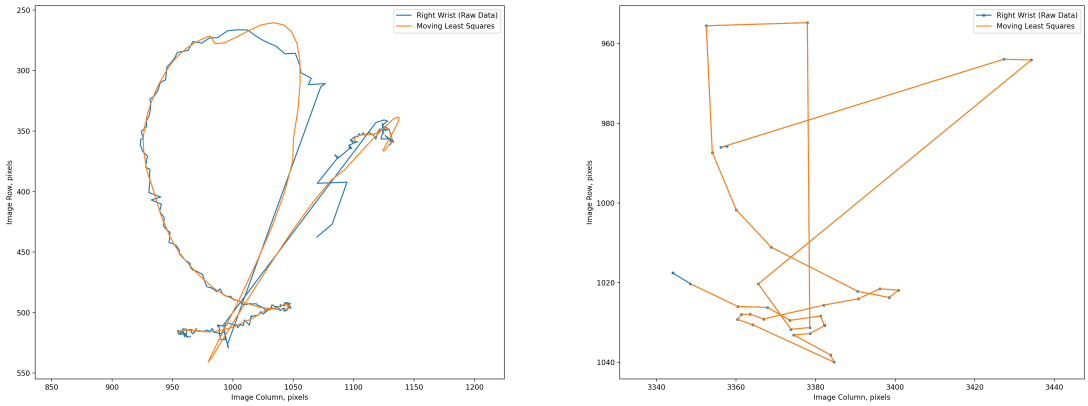
Besides not using any markers for player tracking, the presented system also demonstrates that inferences, such as 3D skeletons, which would typically require multiple camera views can also be achieved from a single camera view (Section 6.3.1). However, it should be understood that such inferences are not 100 % reliable due to the ambiguity in problem formulation (it is an ill-posed optimization problem) and should only be used when appropriate.

One of the major challenges is to understand when and how/why the algorithms such as object detection or pose estimation fail. The output of these algorithms is used to guide data acquisition and processing, so they have to be fast and reliable. Figure 6.22 is showing an example of a pitch during which the pose detection algorithm failed. The mode of failure in this example is a temporary flip between the left and right joints which renders projected speed calculations incorrect and, thus, unusable. It is apparent that the reason for such a failure (invariant to the camera resolution) is the occlusion of the right arm during the most crucial portion

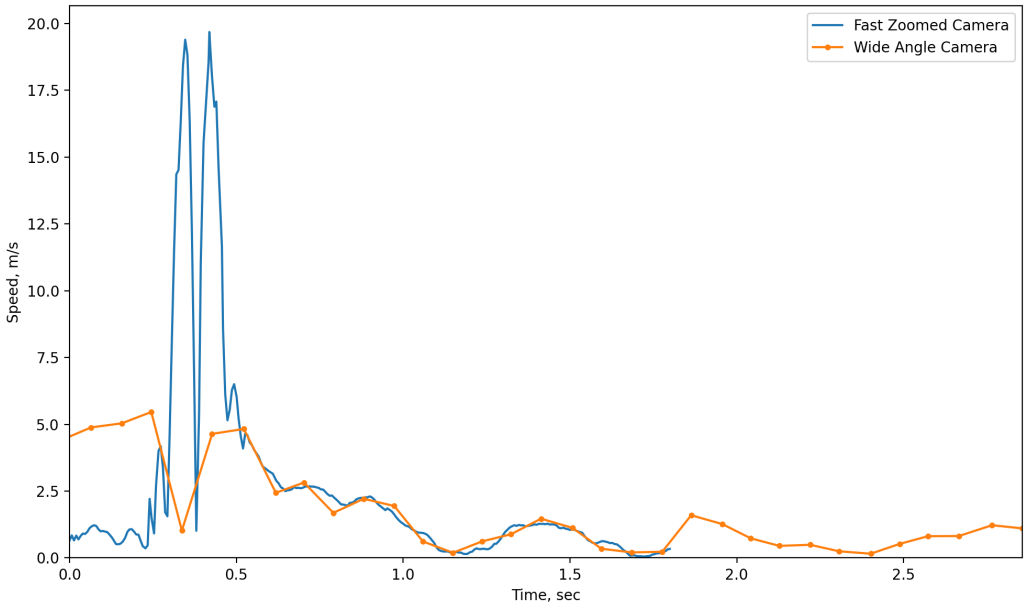
of the pitch movement. A possible way to tackle such occlusion cases is to develop pose estimation algorithms that are not only providing the best estimate of the person's pose but also include confidence level for each joint so that only unoccluded confident detections are being used for downstream analysis.

In contrast to the first and second LegoTracker prototypes, the design of our third prototype did include an entire linear array of four I2S microphones for audio recording with a goal of beamforming-based sound source localization. However, we were unable to recover the synchronization signal when using the I2S protocol with MCHStreamer. The microphones would need to be replaced by a PDM version or an entirely different audio interface should be used to fix this issue, so we excluded audio modality during the field testing of the third prototype.

(**a**) Video Frames



(**b**) Wrist Trajectories



(**c**) Projected Speed

Figure 6.22: Pitch example 3 as recorded by zoomed (left) and wide (right) cameras.

6.6.2 Infrastructure

The LegoTracker sensor described in this chapter is a fairly complex device. In the course of working on this project, we found the need to build a new lab infrastructure in the Visualization Imaging and Data Analytics (VIDA) center at the NYU Tandon School of Engineering (Figure 6.23). It includes the following equipment and machinery:

- Ultimaker S5 3D Printer with Air Manager
- Epilog Mini 12 x 24" 40 Watt Laser Engraver
- Bantam Tools PCB Milling Machine
- Bantam Tools Desktop CNC Milling Machine
- WEN 33075 11-Amp Variable Speed 16-Inch Benchtop Milling Machine
- WEN 3970T 4-inch x 6-inch Band Saw and TackLife 10" Miter Saw
- Spray Painting Setup and Standard Tools (Drills, Files, Screwdrivers, etc.)
- SIGLENT SDS1204X-E 4-Channel 200 MHz Oscilloscope (1 GS/s)
- Soldering Station, Infrared Owen, and Air Tools (Compressor, Dispenser, etc.)
- Ouster OS1 LIDAR Sensor
- Structured Light Scanner and Lighting System

The lab has been funded by the National Science Foundation, NASA and NYU. All these tools were used and necessary for the manufacturing of the LegoTracker prototypes.

Figure 6.23: Lab infrastructure that was developed to make the application possible.

Chapter 7

Conclusions

Sensor networks have dynamically expanded our ability to monitor and study the world. Their presence and need keep increasing, and new hardware configurations expand the range of physical stimuli that can be accurately recorded. However, implementing sensor networks can end up being an enormous engineering feat and often takes away valuable time from the actual research that the researchers are seeking to perform. The increasing availability of maker spaces that are providing 3D printing and other technologies makes the building of new sensors within reach of a broader community, even if still very complex. We have introduced REIP, a Reconfigurable Environmental Intelligence Platform that facilitates and streamlines the process of environmental sensing deployments, providing researchers of varying experience levels with tools and best practices for designing and building sensor networks. We have also built a software framework (an SDK) to make it quicker and easier to prototype and deploy sensor networks and we have shown its use in two different case studies as well as three real-world application examples, demonstrating its utility and versatility.

The first case study in Section 3.4 highlights the use of the REIP SDK in the design of an application pipeline for a multimodal smart filter for urban accident data collection, including near-accident situations involving pedestrians and bicyclists. A finding from this case study was that the REIP workflow (see Section 3.1) allows the sensor architect to design the data pipeline from an abstract, top-down view and have it translate directly into software components without the extraneous complications caused by software and hardware-specific details. In the second case study (Section 3.5), we continued to expand the capabilities of the sensor network for localization and tracking tasks by leveraging dozens of blocks readily available in the REIP SDK, and by incorporating a custom synchronization solution (Section 3.5.3.2).

Lessons learned from the real-world application examples are as follows:

HVAC Systems

An investigation of the correlation between the performance of the heating, ventilation, and air conditioning (HVAC) system in the indoor spaces and actual occupancy of these spaces was conducted to provide insights into building use patterns for adaptive control strategies of the HVAC system with a goal of reducing building's energy consumption. It demonstrated the feasibility of live estimation of indoor spaces occupancy using the REIP platform, which could be used in dynamically controllable HVAC systems for even better performance and energy efficiency. The ability of live data processing using REIP SDK has also enabled anonymity necessary for such a study to be conducted in public spaces (Section 4.5.1).

Urban Dataset

An urban dataset was acquired using multimodal REIP sensors that were built with the support of accurate synchronization. We have shown its utility for

pedestrian-vehicle interaction analysis or algorithm development in the context of smart cities where sensor networks empowered by AI techniques provide a real-time understanding of the environment for different vehicles, including self-driving cars. This application highlights the challenges of urban sensing, such as extreme operating conditions, data loss, or the need for cross-sensor synchronization, and which REIP (SDK) was able to provide solutions for (Section 5.3). It is also important to note that the REIP SDK can handle data of varying structure and sizes, and does not add any significant overhead (Section 3.3.2.2), so we were able to easily include new modalities (i.e. LIDAR) and maximize the utilization of available computational resources.

Sports Tracking

Sports tracking systems revolutionized sports analytics and the way coaches manage players and approach the game. However, for past decades sports tracking was limited to a rough representation of each player by a single point and often relies on special markers integrated into sports apparel. Recent advances in deep learning and computer vision algorithms enabled markerless detection of human pose. In this application, we have presented a novel modular sports tracking system (Section 6.2) providing a significantly higher level of detail in game tracking. Comprising of independent units, each running state-of-the-art algorithms for player detection and tracking, it provides a full skeleton representation for each player over a large game field as well as high-level game events with precise timing. We have gone through three iterations of the hardware prototypes of the tracking units (Sections 6.3, 6.4, and 6.5) and performed an in-field testing.

Advanced data serialization and block connection strategies (Sections 3.3.1.2 and 3.2.3) offered by the REIP SDK have been essential for this application. They en-

abled us to find the right balance between multi-threading and multi-processing for optimal performance, which is crucial for real-time data processing during high frame rate video recording. Without the live feedback from the pipeline, it would be impossible to keep the players within the field of view of our high-speed zoomed cameras and achieve an order of magnitude improvement in signal-to-noise ratio (Section 6.6).

7.1 Limitations and Future Work

We have learned the following lessons with regard to REIP's shortcomings as a result of three real-world application examples performed by users of various backgrounds and levels of expertise.

First is that documentation in a form of complete examples is far more effective. Master's students implementing the HVAC application were able to extend the sensor's capabilities to environmental sensing starting off the existing examples (Section 3.2.5). But it turned out to be harder for the more experienced students acquiring the urban dataset to fully leverage the timing metadata provided by REIP blocks for synchronization purposes. Therefore, increasing the number of fully documented application examples is important for improving the REIP's accessibility.

One of the primary focuses during the design and development of the REIP SDK was the minimization of its overhead (Section 3.3.2) so that users can maximize the performance of their data acquisition and processing pipelines. We believe to have achieved this objective as demonstrated by the sports tracking application involving high-performance units with real-time feedback loops. However, despite the high optimization on the individual sensor design level, we have found the system to be lacking when operating a fleet of multiple sensors. More tools are necessary for

managing deployments of a larger number of sensors, including a live dashboard of sensors' status to maximize the uptime during prolonged data acquisition sessions or permanent deployments.

This work represents the first step towards building a large ecosystem of tools that make it faster and easier to build and deploy (wireless) sensor networks. To further increase the usability and accessibility of the REIP platform, we are planning to develop a Graphical User Interface (GUI) for application pipeline creation using the Node-RED [90] platform. This browser-based interface would provide visual representations of the functional blocks, which could then be used to wire-up application pipelines by less technical users. The GUI could take into account the constraints of the proposed system, as well as allow users to define custom application-specific hardware/software constraints. The resulting application pipeline would then be exported in the form of a deployable script that is ready to be executed on sensor nodes with our run-time installed.

Hardware and software integration poses a significant challenge in remote sensing, which is subject to a number of constraints including computing resources available, hardware I/O offered, sensing options, inter-process data rates, and available remote connectivity options. With the application pipeline defined using the REIP SDK, this integration process becomes less of a challenge, as the blocks chosen dictate the minimal hardware platform that can support it. The presented process of manually benchmarking possible hardware platforms (Section 3.3.2.2) is not ideal but is a precursor to our planned simulation and optimization tool for pipeline evaluation, where optimal hardware platforms would be matched to an application pipeline in an automatic way subject to user constraints, such as maximum memory usage, data output rate, etc.

As mentioned in Chapter 5, we are also currently working on a more comprehensive analysis of the urban dataset using all of the available data modalities. Our objective is to provide examples of analysis that would not otherwise be possible without the unique qualities of our dataset, such as multimodality, multiview and high resolution with precise synchronization. One example of such analysis is the evaluation of pedestrians' movement per traffic light cycle. The data will be anonymized to address privacy concerns and ensure "intelligence without surveillance".

The LegoTracker sports tracking system is in need of further testing and development. In particular, we intend to develop better calibration procedures for joint analysis of the data captured by different tracking units. Additionally, we are exploring ways of making the control software needed for managing the larger number of tracking units compatible with our vision of a more generic REIP GUI tool.

Bibliography

[1] Iyad Kheirbek, Kazuhiko Ito, Richard Neitzel, Jung Kim, Sarah Johnson, Zev Ross, Holger Eisl, and Thomas Matte. Spatial variation in environmental noise and air pollution in new york city. *Journal of Urban Health*, 91(3):415–431, 2014.

[2] Juan P Bello, Claudio Silva, Oded Nov, R Luke Dubois, Anish Arora, Justin Salamon, Charles Mydlarz, and Harish Doraiswamy. Sonyc: A system for monitoring, analyzing, and mitigating urban noise pollution. *Communications of the ACM*, 62(2):68–77, 2019.

[3] Paolo Bellagente, Paolo Ferrari, Alessandra Flammini, and Stefano Rinaldi. Adopting iot framework for energy management of smart building: A real test-case. In *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 138–143. IEEE, 2015.

[4] NVIDIA AGX. NVIDIA AGX - jetson agx xavier developer kit. `developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit/`, 2021. [Online; accessed 02-Jun-2021].

[5] NVIDIA. Jetson - Platform for ai at the edge. `developer.nvidia.com/embedded-computing`, 2021.

[6] RaspberryPi. RaspberryPi - raspberry pi 400 computer kit. `raspberrypi.org//`, 2021. [Online; accessed 02-Jun-2021].

[7] Yurii Piadyk, Bea Steers, Charlie Mydlarz, Mahin Salman, Magdalena Fuentes, Junaid Khan, Hong Jiang, Kaan Ozbay, Juan Pablo Bello, and Claudio Silva. Reip: A reconfigurable environmental intelligence platform and software framework for fast sensor network prototyping. *Sensors*, 22(10), 2022.

[8] Somansh Kumar and Ashish Jasuja. Air quality monitoring system based on iot using raspberry pi. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1341–1346. IEEE, 2017.

[9] Manish Kushwaha, Songhwai Oh, Isaac Amundson, Xenofon Koutsoukos, and Akos Ledeczi. Target tracking in heterogeneous sensor networks using audio and video sensor fusion. In *2008 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 14–19. IEEE, 2008.

[10] Marcos Lage, Jorge Piazentin Ono, Daniel Cervone, Justin Chiang, Carlos Dietrich, and Claudio T. Silva. Statcast dashboard: Exploration of spatiotemporal baseball data. *IEEE Computer Graphics and Applications*, 36(5):28–37, 2016.

[11] Anand Nayyar and Rajeshwar Singh. A comprehensive review of simulation tools for wireless sensor networks (wsns). *Journal of Wireless Networking and Communications*, 5(1):19–47, 2015.

[12] Alexander Gluhak, Srdjan Krco, Michele Nati, Dennis Pfisterer, Nathalie Mitton, and Tahiry Razafindralambo. A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, 49(11):58–67, 2011.

[13] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, et al. Fit iot-lab: A large scale open experimental iot testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464. IEEE, 2015.

[14] Rachit Agarwal, David Gomez Fernandez, Tarek Elsaleh, Amelie Gyrard, Jorge Lanza, Luis Sanchez, Nikolaos Georgantas, and Valerie Issarny. Unified iot ontology to enable interoperability and federation of testbeds. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 70–75. IEEE, 2016.

[15] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. The signpost platform for city-scale sensing. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '18, page 188–199. IEEE Press, 2018.

[16] Joseph Rafferty, Jonathan Synnott, Andrew Ennis, Chris Nugent, Ian McChesney, and Ian Cleland. Sensorcentral: A research oriented, device agnostic, sensor data platform. In *International conference on ubiquitous computing and ambient intelligence*, pages 97–108. Springer, 2017.

[17] Charles E Catlett, Peter H Beckman, Rajesh Sankaran, and Kate Kusiak

Galvin. Array of things: a scientific research instrument in the public way: platform design and early lessons learned. In *Proceedings of the 2nd international workshop on science of smart city operations and platforms engineering*, pages 26–33, 2017.

[18] Sage project. `sagecontinuum.org`, 2020. [Online; accessed October 26, 2020].

[19] Libelium. Libelium - waspmote frame library. `development.libelium.com/data-frame-programming-guide/introduction/`, 2021. [Online; accessed 02-Jun-2021].

[20] Usc testbed. `cci.usc.edu/index.php/cci-iot-testbed`, 2020. [Online; accessed October 24, 2020].

[21] FIWARE. FIWARE - Open source software platform components. `fiware.org/developers/catalogue//`, 2021. [Online; accessed 01-Jun-2021].

[22] Joseph Noor, Sandeep Singh Sandha, Luis Garcia, and Mani Srivastava. Ddflow visualized declarative programming for heterogeneous iot networks on heliot testbed platform: Demo abstract. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, page 287–288, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Borui Li and Wei Dong. Edgeprog: Edge-centric programming for iot applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 212–222. IEEE, 2020.

[24] Xiaochen Liu, Pradipta Ghosh, Oytun Ulutan, B. S. Manjunath, Kevin Chan, and Ramesh Govindan. Caesar: Cross-camera complex activity recognition.

In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 232–244, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Pete Beckman, Rajesh Sankaran, Charlie Catlett, Nicola Ferrier, Robert Jacob, and Michael Papka. Waggle: An open sensor platform for edge computing. In *2016 IEEE SENSORS*, pages 1–3. IEEE, 2016.

[26] Waggle. Waggle - Open platform for ai@edge computing and intelligent sensors. `wa8.gl/code-docs/`, 2021. [Online; accessed 01-Jun-2021].

[27] Apache Ray. Apache Ray - Fast and simple distributed computing. `ray.io/`, 2021. [Online; accessed 01-Jun-2021].

[28] Celery. Celery - distributed task queue. `docs.celeryproject.org/en/stable/index.html/`, 2021. [Online; accessed 02-Jun-2021].

[29] Luigi. Luigi - workflow management pipeline. `luigi.readthedocs.io/en/stable/`, 2021. [Online; accessed 01-Jun-2021].

[30] GStreamer. GStreamer - Open source multimedia framework. `gstreamer.freedesktop.org`, 2021. [Online; accessed 01-Jun-2021].

[31] NVIDIA DeepStream. NVIDIA DeepStream - deepstream sdk ai powered intelligent video analytics. `developer.nvidia.com/deepstream-sdk/`, 2021. [Online; accessed 02-Jun-2021].

[32] FFmpeg. FFmpeg - Cross platform solution for audio and video. `ffmpeg.org/`, 2021. [Online; accessed 01-Jun-2021].

[33] Congduc Pham. Communication performances of ieee 802.15. 4 wireless sensor motes for data-intensive applications: A comparison of waspmote, arduino mega, telosb, micaz and imote2 for image surveillance. *Journal of Network and Computer Applications*, 46:48–59, 2014.

[34] Apache Airflow. Apache Airflow - opensource platform. `airflow.apache.org/docs/`, 2021. [Online; accessed 02-Jun-2021].

[35] Aigerim Zhalgasbekova, Arkady Zaslavsky, Saguna Saguna, Karan Mitra, and Prem Prakash Jayaraman. Opportunistic data collection for iot-based indoor air quality monitoring. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 53–65. Springer, 2017.

[36] Alexey Medvedev, Alireza Hassani, Arkady Zaslavsky, Prem Prakash Jayaraman, Maria Indrawan-Santiago, Pari Delir Haghighi, and Sea Ling. Data ingestion and storage performance of iot platforms: study of openiot. In *International Workshop on Interoperability and Open-Source Solutions*, pages 141–157. Springer, 2016.

[37] Peter Salhofer and FH Joanneum. Evaluating the fiware platform: A case-study on implementing smart application with fiware. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, volume 9, pages 5797–5805, 2018.

[38] Victor Araujo, Karan Mitra, Saguna Saguna, and Christer Åhlund. Performance evaluation of fiware: A cloud-based iot platform for smart cities. *Journal of Parallel and Distributed Computing*, 132:250–261, 2019.

[39] Joseph Noor. Ddflow. `https://github.com/nesl/DDFlow`, 2020.

[40] Apache Spark. Apache Spark - Unified analytics engine for large-scale data processing. `spark.apache.org/`, 2021. [Online; accessed 01-Jun-2021].

[41] Md Mahbub Alam, Suprio Ray, and Virendra C. Bhavsar. A performance study of big spatial data systems. In *Proceedings of the 7th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial 2018, page 1–9, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Kasumi Kato, Atsuko Takefusa, Hidemoto Nakada, and Masato Oguchi. A study of a scalable distributed stream processing infrastructure using ray and apache kafka. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5351–5353. IEEE, 2018.

[43] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H Peter Hofstee, and Zaid Al-Ars. Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In *International Symposium on Applied Reconfigurable Computing*, pages 32–47. Springer, 2019.

[44] Tanveer Ahmad, Nauman Ahmed, Zaid Al-Ars, and H Peter Hofstee. Optimizing performance of gatk workflows using apache arrow in-memory data framework. *BMC genomics*, 21(10):1–14, 2020.

[45] Geoffrey Lentner. Shared memory high throughput computing with apache arrow™. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Xintian Wu, Pengfei Qu, Shaofei Wang, Lin Xie, and Jie Dong. Extend the

ffmpeg framework to analyze media content. *arXiv preprint arXiv:2103.03539*, 2021.

[47] Francois Chollet et al. Keras. `github.com/fchollet/keras`, 2015. [Online; accessed on 2 June 2021].

[48] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Transactions on Graphics*, 37(4):1–13, Aug 2018.

[49] Apache Arrow Plasma. Apache Arrow Plasma - The Plasma In-Memory Object Store. `minidsp.com/products/usb-audio-interface/mchstreamer`, 2021. [Online; accessed 04-Jun-2021].

[50] C. Mydlarz, M. Sharma, Y. Lockerman, B. Steers, C. Silva, and J. P. Bello. The life of a new york city noise sensor network. *Sensors*, 19(6):1415, 2019.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[52] miniDSP. miniDSP - MCHStreamer Kit. `minidsp.com/products/usb-audio-interface/mchstreamer`, 2021. [Online; accessed 03-Jun-2021].

[53] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh

Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[54] GStreamer – Open source multimedia framework. `gstreamer.freedesktop.org`, 2020. [Online; accessed on 1 June 2021].

[55] Robin Scheibler, Eric Bezzam, and Ivan Dokmanic. Pyroomacoustics: A python package for audio room simulations and array processing algorithms. *CoRR*, abs/1710.04196, 2017.

[56] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[57] M. Devendra and K. Manjunathachari. Doa estimation of a system using music method. In *2015 International Conference on Signal Processing and Communication Engineering Systems*, pages 309–313, 2015.

[58] Energy Information Administration. Electric Power Annual. Technical report, U.S., 2017.

[59] U.S. Department of Energy. Energy Efficiency Trends in Residential and Commercial Buildings. *Energy*, pages 1–32, 2008.

[60] Mohamad Fadzli Haniff, Hazlina Selamat, Rubiyah Yusof, Salinda Buyamin, and Fatimah Sham Ismail. Review of HVAC scheduling techniques for build-

ings towards energy-efficient and cost-effective operations. *Renewable and Sustainable Energy Reviews*, 27:94–103, 2013.

[61] G. Escrivá-Escrivá, I. Segura-Heras, and M. Alcázar-Ortega. Application of an energy management and control system to assess the potential of different control strategies in HVAC systems. *Energy and Buildings*, 42(11):2258–2267, 2010.

[62] K. J. Chua, S. K. Chou, W. M. Yang, and J. Yan. Achieving better energy-efficient air conditioning - A review of technologies and strategies, 2013.

[63] Jiakang Lu, Tamim Sookoor, Vijay Srinivasan, Ge Gao, Brian Holben, John Stankovic, Eric Field, and Kamin Whitehouse. The Smart Thermostat: Using Occupancy Sensors to Save Energy in Homes. *Proceedings of ACM SenSys*, 55:211–224, 2010.

[64] Varick L. Erickson, Miguel a. Carreira-Perpinan, and Alberto E. Cerpa. OBSERVE: Occupancy-based system for efficient reduction of HVAC energy. *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 258–269, 2011.

[65] Ali Ghahramani, Guillermo Castro, Burcin Becerik-Gerber, and Xinran Yu. Infrared thermography of human face for monitoring thermoregulation performance and estimating personal thermal comfort. *Building and Environment*, 109:1–11, 2016.

[66] World Business Council EEB. Report. Technical report, U.S., 2016.

[67] Wei Wang, Jiayu Chen, Gongsheng Huang, and Yujie Lu. Energy efficient

HVAC control for an IPS-enabled large space in commercial buildings through dynamic spatial occupancy distribution. *Applied Energy*, 207:305–323, 2017.

[68] Tianzhen Hong, Sarah C. Taylor-Lange, Simona D'Oca, Da Yan, and Stefano P. Corgnati. Advances in research and applications of energy-related occupant behavior in buildings. *Energy and Buildings*, 116:694–702, 2016.

[69] H Sachs. Programmable thermostats. *American Council for an Energy Efficient Economy(ACEEE)*, 2004.

[70] Robert J. Meyers, Eric D. Williams, and H. Scott Matthews. Scoping the potential of monitoring and control technologies to reduce energy use in homes. *Energy and Buildings*, 42(5):563–569, 2010.

[71] Texas Instruments. Energia IDE. `https://energia.nu/`, 2021. [Online; accessed 01-Jun-2021].

[72] Department of the Environment and Energy. Hvac energy breakdown. *Heating, Ventilation and Air-Conditioning High Efficiency Systems Strategy*, pages 1–2, 2013.

[73] NYU Office of Sustainability. Nyu climate action plan update 2021. -, 20.

[74] Yiying Wu, Xianghua(Sharon) Ding, Xuelan Dai, Peng Zhang, Tun Lu, and Ning Gu. Alignment work for urban accessibility: A study of how wheelchair users travel in urban spaces. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), nov 2022.

[75] Filip Biljecki and Koichi Ito. Street view imagery in urban analytics and gis: A review. *Landscape and Urban Planning*, 215:104217, 2021.

[76] Bowen Cheng, Bin Xiao, Jingdong Wang, Honghui Shi, Thomas S. Huang, and Lei Zhang. Higherhrnet: Scale-aware representation learning for bottom-up human pose estimation, 2019.

[77] G. Pingali, A. Opalach, Y. Jean, and I. Carlbom. Visualization of sports using motion trajectories: providing insights into performance, style, and strategy. In *Proceedings Visualization, 2001. VIS '01.*, pages 75–544, 2001.

[78] Rui Yuan, Zhendong Zhang, Pengwei Song, Jia Zhang, and Long Qin. Construction of virtual video scene and its visualization during sports training. *IEEE Access*, 8:124999–125012, 2020.

[79] Major League Baseball Advanced Media (MLBAM). Statcast. URL: `http://m.mlb.com/glossary/statcast`, March 2015.

[80] NBA and Stats LLC. NBA partners with Stats LLC for tracking technology. URL: `https://www.stats.com/sportvu-basketball/`, September 2013.

[81] Mark Hedley, Colin Mackintosh, Richard Shuttleworth, David Humphrey, Thuraiappah Sathyan, and Phil Ho. Wireless tracking system for sports training indoors and outdoors. *Procedia Engineering*, 2(2):2999–3004, 2010. The Engineering of Sport 8 - Engineering Emotion.

[82] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *Lecture Notes in Computer Science*, page 818–833, 2014.

[83] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks, 2013.

[84] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos, 2014.

[85] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[86] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[87] Y. C. Chiu, C. Y. Tsai, M. D. Ruan, G. Y. Shen, and T. T. Lee. Mobilenet-ssdv2: An improved object detection model for embedded systems. In *2020 International Conference on System Science and Engineering (ICSSE)*, pages 1–5, 2020.

[88] CMU. Moption capture. `http://mocap.cs.cmu.edu`, 2021. [Online; accessed on 01-Jun-2021].

[89] Wah Yen Tey, Nor Azwadi Che Sidik, Nor Azwadi Che Sidik, Yutaka Asako, Mohammed Muhieldeen, Omid Afshar, and W Tey. Moving least squares method and its improvement: A concise review. *Journal of Applied and Computational Mechanics*, 7:883–889, 04 2021.

[90] NodeRED. NodeRED - Low-code programming for event-driven applications. `nodered.org/`, 2021. [Online; accessed 01-Jun-2021].