Universidad de Valladolid

Escuela de Ingeniería Informática

Grado en Ingeniería Informática
Mención De Tecnologías de la Información

TRABAJO FIN DE GRADO

Adquisición de mediciones físicas en tiempo real provenientes de sensores MEMS sobre una raqueta de badminton

Autor:
Nicolás Martín García

Tutor:
Dr. César Llamas Bello

Resumen

La introducción de sensores del tipo MEMS en la monitorización de actividades físicas abre las puertas a la innovación e investigación en la biomecánica propia de las actividades deportivas, especialmente aquellas donde se emplean elementos como raquetas u otros dispositivos de mano. Estos sensores permiten obtener mediciones de aceleraci´on, velocidad angular y orientación, lo que facilita el análisis biomecánico de los movimientos del jugador durante su actividad.

Este Trabajo Fin de Grado presenta el desarrollo de un sistema que captura, procesa y visualiza datos obtenidos en tiempo real mediante sensores MEMS integrados en la empuñadura de una raqueta de bádminton. El sistema emplea tecnologías inalámbricas para enviar los datos a una base formada por una computadora que puede actuar autónomamente.

El proyecto realizado se estructura en varias fases, desde la integración de una placa Arduino Nano IoT 33 en la raqueta, hasta la implementaci´on de una interfaz web interactiva para la visualización de los datos. La captura de datos se realiza en tiempo real y es procesada por un servidor Flask, que permite almacenar los resultados para un análisis posterior.

Se ha implementado un sistema robusto con varias modalidades de captura de datos, incluyendo la captura continua, temporizada y pruebas controladas de 10 segundos para verificar la funcionalidad de los sensores. Los datos se visualizan mediante gráficos interactivos, con opción de almacenamiento en archivos CSV para su uso posterior.

Palabras clave: Sensores MEMS, Arduino Nano IoT, Flask, Captura de datos en tiempo real, Bádminton.

Abstract

The integration of MEMS sensors in the monitoring of physical activities fosters innovation and research in sports biomechanics, particularly in sports utilizing rackets or handheld devices. These sensors enable real-time measurements of acceleration, angular velocity, and orientation, providing valuable insights into players' biomechanical movements during activity.

This Final Degree Project presents the development of a system that captures, processes, and visualizes real-time data collected through MEMS sensors embedded in the handle of a badminton racket. The system uses wireless technologies to transmit data to a computing unit that can operate autonomously.

The project is structured in multiple stages, including the integration of an Arduino Nano IoT 33 board into the racket and the development of an interactive web interface for data visualization. Real-time data capture is processed by a Flask server, which also facilitates data storage for further analysis.

A robust system has been implemented with various data capture modes, including continuous capture, timed capture, and controlled 10-second tests to verify sensor functionality. Data is displayed through interactive graphs and can be stored in CSV files for future use.

Key words: MEMS sensors, Arduino Nano IoT, Flask, Real-time data capture, Badminton.

Agradecimientos

A mi familia y amigos, quienes han sido mi mayor fuente de fortaleza a lo largo de estos meses. Gracias por ser un refugio de motivación en los momentos más complejos y por no dejarme caer cuando sentía que no podía más. En especial, a Lorena, por estar siempre a mi lado con su cariño y apoyo incondicional. Sin ti, este proyecto no hubiera sido posible de la misma manera.

Me gustaría expresar mi más sincero agradecimiento a mi tutor, César Llamas Bello, por su constante guía y apoyo durante todo el proyecto. Su paciencia, conocimiento y orientación fueron fundamentales, especialmente en los momentos en los que el camino parecía más difícil. Gracias a su acompañamiento, hemos podido superar juntos los desafíos que se presentaron y alcanzar los objetivos que me había propuesto.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

Capítulo 1

Introducción

1.1. Contexto y motivación

El bádminton es un deporte que, a pesar de su aparente simplicidad, involucra una combinación de movimientos rápidos, agudos y repetitivos que pueden afectar significativamente el cuerpo del jugador. Durante un partido, los jugadores experimentan fuerzas intensas y aceleraciones rápidas que impactan en sus articulaciones, músculos y tendones. Comprender con exactitud cómo estos movimientos afectan al cuerpo, y específicamente cómo influyen los golpes realizados con la raqueta, sigue siendo un desafío para entrenadores y deportistas.

El uso de tecnología de adquisición de datos mediante sensores MEMS (Microelectromechanical Systems) en una raqueta de bádminton permite obtener mediciones físicas precisas en tiempo real de parámetros clave, como la aceleración, la velocidad de los golpes y la orientación de la raqueta. Esta información puede ofrecer una visión detallada sobre el rendimiento del jugador y las dinámicas que ocurren durante el juego. Las aplicaciones de esta tecnología no solo permiten analizar el comportamiento de la raqueta, sino también estudiar cómo los movimientos y fuerzas aplicadas afectan el cuerpo del jugador, ayudando a identificar posibles riesgos de lesiones, optimizar técnicas de golpeo y mejorar la eficiencia del entrenamiento.

En el ámbito del rendimiento deportivo, contar con datos en tiempo real abre la puerta a nuevas oportunidades de personalización y análisis. Los entrenadores pueden obtener información más precisa sobre la biomecánica del jugador, lo que permite ajustar las estrategias de entrenamiento en función de la técnica individual. Además, esta tecnología puede ser utilizada para prevenir lesiones recurrentes, corregir posturas inadecuadas y mejorar el rendimiento a largo plazo.

El desarrollo de un sistema de adquisición de datos aplicado al bádminton, usando sensores MEMS en la raqueta, representa un avance tecnológico significativo, no solo en el ámbito deportivo, sino también en el análisis biomecánico del cuerpo humano durante actividades físicas intensas.

Este Trabajo Fin de Grado propone el desarrollo de un sistema capaz de capturar, procesar y visualizar datos obtenidos en tiempo real mediante sensores MEMS integrados en la empuñadura de una raqueta de bádminton. Este sistema utiliza tecnologías inalámbricas para transmitir los datos a una computadora base que puede operar de forma autónoma, permitiendo posteriormente el envío de la información a un repositorio en la nube con el mínimo esfuerzo.

El proyecto se organiza en varias fases, comenzando con la integración de una placa Arduino Nano IoT 33 en la raqueta y culminando con la implementación de una interfaz web interactiva para la visualización de los datos. La captura se realiza en tiempo real, y un servidor Flask se encarga del procesamiento de la información y del almacenamiento de los resultados para su análisis posterior.

1.2. Objetivos

Generales:

- Integrar en el mango de la raqueta de bádminton un sistema de sensorización inalámbrico que disponga de una velocidad suficiente de adquisición.
- Realizar un sistema hardware/software que permita el control del dispositivo integrado y la descarga de datos de la medición para su posterior volcado y análisis.

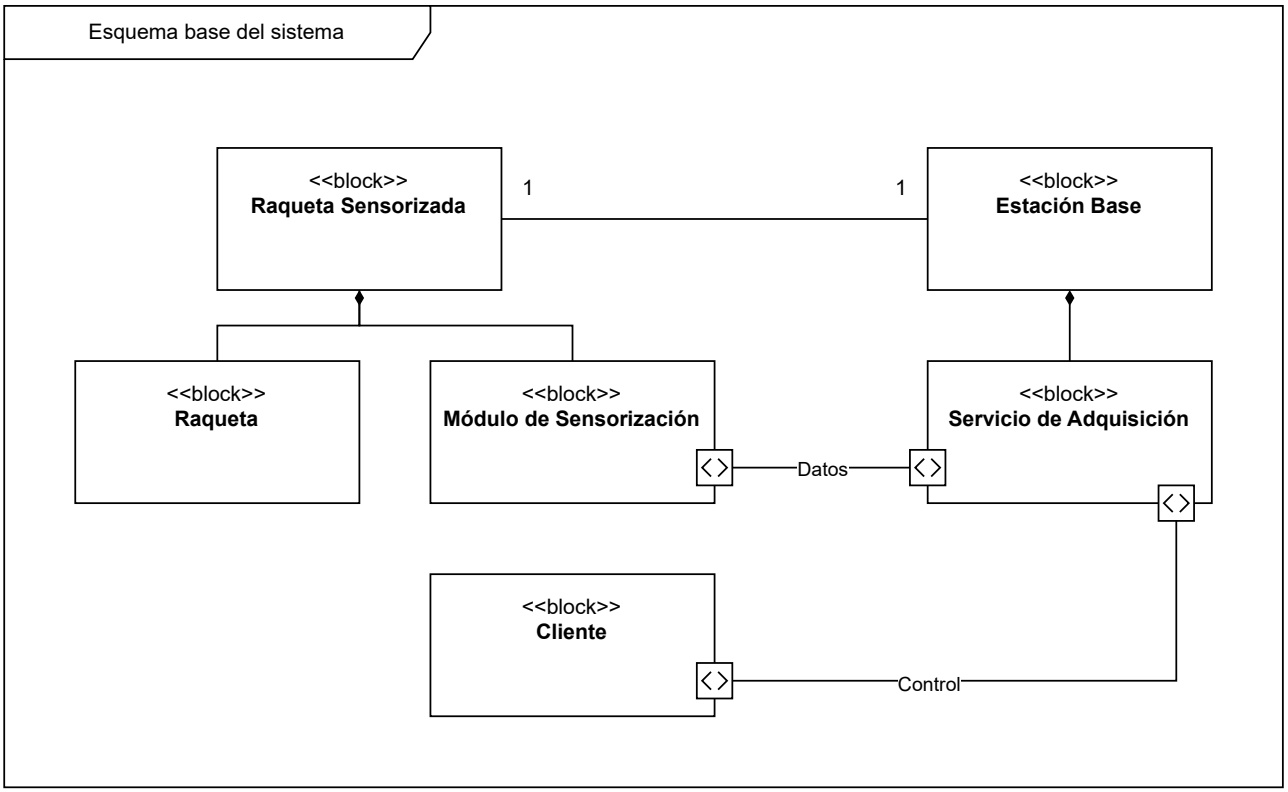

Figura 1.1: Arquitectura Base

Específicos:

- Integrar y programar una placa Arduino Nano IoT 33 con capacidades inalámbricas y sensores MEMS (acelerómetros, giroscopios, etc.) en una raqueta de bádminton para medir parámetros como aceleración, velocidad angular y orientación durante el juego.

- Desarrollar una plataforma de adquisición y procesamiento de datos en tiempo real utilizando la Arduino Nano IoT 33 y un servidor Flask, que permita la captura, procesamiento y transmisión de los datos a través de distintas modalidades:
  - Toma de datos continua.
  - Toma de datos por un tiempo determinado.
- Implementar una interfaz de visualización de datos en el momento, que permita mostrar gráficamente los resultados obtenidos desde la Arduino Nano IoT 33 y el servidor Flask, facilitando su análisis para evaluar el rendimiento y la técnica de los jugadores.
- Explorar la posibilidad de analizar los datos recopilados para identificar patrones de movimientos y golpes, así como su impacto en el rendimiento del jugador y el potencial riesgo de lesiones.
- Validar la precisión del sistema mediante pruebas experimentales controladas, utilizando las herramientas del servidor Flask para verificar que los datos capturados se correspondan con la realidad, y guardarlos en caso de ser correctos.
- Proponer mejoras técnicas en función de los resultados obtenidos, optimizando el sistema tanto para el análisis del juego como para su aplicación en la mejora del rendimiento y la prevención de lesiones.

1.3. Estructura del documento

La estructura de la memoria del presente Trabajo de Fin de Grado (TFG) se corresponde con los siguientes capítulos:

1. **Introducción**

   La introducción establece el contexto general del proyecto, exponiendo las motivaciones que lo impulsan. Asimismo, se detallan los objetivos generales y específicos, y se proporciona una visión global de la estructura del documento, facilitando una guía para su lectura.

2. **Planificación del Proyecto**

   Este capítulo define las etapas y plazos del proyecto para asegurar su desarrollo eficiente. Se describen las fases principales, como la investigación, el diseño, la implementación y las pruebas, asignando tiempos estimados para cada una.

3. **Fundamentación Teórica**

   Este capítulo ofrece una revisión de los conceptos teóricos necesarios para comprender el desarrollo del trabajo. Se proporciona un marco conceptual que sienta las bases para el análisis y diseño posterior del sistema.

4. **Análisis y diseño**

   En este apartado se detallan los requisitos funcionales y no funcionales del sistema, así como la descripción de los casos de uso más relevantes. Se incluye un diagrama de arquitectura que presenta la estructura global del sistema y se complementa con diagramas detallados, como los diagramas de clases, secuencia y actividades, que ilustran el diseño específico.

5. **Implementación**

   Se describe minuciosamente el proceso de desarrollo del sistema, especificando el entorno de desarrollo, las herramientas utilizadas y el detalle de los módulos implementados.

6. **Resultados y discusión**

   Este capítulo expone los resultados obtenidos tras la implementación del sistema y realiza un análisis crítico de los mismos, comparándolos con los objetivos iniciales. También se discuten las posibles limitaciones encontradas y los desafíos afrontados durante el desarrollo.

7. **Conclusiones y líneas futuras**

   Se realiza un resumen de las principales contribuciones del trabajo, evaluando si los objetivos propuestos han sido alcanzados. Finalmente, se sugieren posibles líneas de investigación o desarrollo futuro que permitan mejorar o continuar el proyecto.

8. **Bibliografía**

   En este capítulo se incluye la lista completa de las fuentes bibliográficas consultadas durante la elaboración del TFG, siguiendo un formato de citación adecuado según los estándares académicos.

9. **Anexos**

   Los anexos incluyen material adicional que complementa el trabajo, como fragmentos de código fuente, manuales de usuario o guías de instalación que puedan ser útiles para comprender mejor el desarrollo e implementación del sistema.

Capítulo 2

Planificación del Proyecto

2.1. Enfoque Metodológico

Para el desarrollo del presente proyecto, se ha seguido un enfoque basado en prototipos iterativos, permitiendo avanzar de manera incremental en la construcción del sistema. Este enfoque es especialmente adecuado para proyectos de naturaleza tecnológica, donde los requisitos pueden evolucionar a medida que se avanza en el desarrollo.

2.1.1. Características principales:

- **Iteraciones definidas**: Cada prototipo tiene un conjunto claro de objetivos que permiten validar el progreso y realizar ajustes necesarios. Esto asegura que cada componente del sistema cumpla con los requisitos establecidos antes de avanzar a la siguiente etapa.
- **Retroalimentación continua**: Tras cada iteración, se revisan los resultados obtenidos con el prototipo, lo que permite identificar problemas o áreas de mejora. Esta retroalimentación se utiliza para refinar el diseño y la implementación en las etapas posteriores.
- **Flexibilidad para adaptarse a cambios**: Al trabajar en iteraciones más pequeñas y manejables, es posible realizar modificaciones sin afectar de manera significativa el desarrollo global del proyecto.
- **Integración final**: Culminado el desarrollo iterativo, se realiza una integración completa del sistema. Esta etapa garantiza que todos los componentes trabajen de manera coherente y efectiva, cumpliendo con los objetivos generales del proyecto.

2.1.2. Ventajas del enfoque iterativo:

1. **Gestión de riesgos**: Permite detectar y solucionar problemas desde las primeras etapas del desarrollo, reduciendo riesgos asociados al diseño, implementación y validación.
2. **Priorización de funcionalidades**: Facilita la identificación de funcionalidades clave que deben estar operativas desde el principio.

3. **Validación incremental**: Los prototipos funcionales permiten realizar pruebas parciales antes de completar el sistema completo, asegurando que cada módulo cumpla con los estándares requeridos.

Este enfoque asegura que cada etapa del proyecto sea gestionada de forma efectiva, reduciendo riesgos y mejorando la calidad del resultado final. Adicionalmente, fomenta la transparencia y la comunicación entre los participantes del proyecto, proporcionando claridad sobre los avances y los objetivos alcanzados en cada iteración.

2.2. Etapas del Proyecto

El desarrollo del proyecto se ha dividido en las siguientes etapas principales:

2.2.1. Análisis de Requisitos

- **Duración**: 2 semanas.
- **Objetivo principal**: Identificar las necesidades funcionales y técnicas del sistema, asegurando que todos los elementos requeridos sean considerados antes de iniciar el diseño.
- **Actividades clave**:

    1. Reuniones iniciales con el tutor y partes interesadas: Definir los objetivos generales del proyecto y las expectativas.
    2. Definición de casos de uso: Describir las interacciones esperadas entre el sistema y el usuario, especificando los flujos principales y alternativos.
    3. Revisión de tecnologías disponibles: Evaluar las herramientas de hardware y software más adecuadas para cumplir con los requisitos definidos.
    4. Estudio de viabilidad técnica: Analizar las posibles limitaciones del proyecto y proponer soluciones que garanticen su factibilidad.

2.2.2. Diseño del Sistema

- **Duración**: 3 semanas.
- **Objetivo principal**: Desarrollar una arquitectura sólida y bien documentada que sirva como guía para la implementación.
- **Actividades clave**:

    1. Diseño de la arquitectura del sistema: Establecer la estructura general, incluyendo módulos principales, flujos de datos y puntos de integración.
    2. Creación de diagramas: Elaborar diagramas de flujo, de clases y de secuencia para detallar la funcionalidad de cada componente.

3. Especificación de protocolos de comunicación: Definir cómo interactuarán los componentes del sistema (e.g., Arduino y servidor Flask).
4. Planificación de pruebas iniciales: Diseñar casos de prueba para validar cada módulo en etapas posteriores.

2.2.3. Desarrollo de Prototipos

- **Duración**: 8 semanas.
- **Objetivo principal**: Construir iterativamente el sistema, incorporando funcionalidades clave en cada prototipo.
- **Iteraciones principales**:
  1. **Prototipo inicial:**
     - Integración de sensores MEMS en la raqueta.
     - Verificación básica de la funcionalidad de los sensores.
     - Implementación inicial de la comunicación inalámbrica.
  2. **Prototipo funcional:**
     - Captura y transmisión básica de datos.
     - Desarrollo de un servidor Flask que reciba los datos y los procese en tiempo real.
     - Pruebas iniciales de visualización gráfica.
  3. **Prototipo avanzado:**
     - Visualización interactiva en tiempo real.
     - Implementación de almacenamiento en archivos CSV.
     - Optimización del rendimiento del sistema.

2.2.4. Pruebas y Validación

- **Duración**: 4 semanas.
- **Objetivo principal**: Garantizar que el sistema cumple con los requisitos definidos y funciona correctamente en escenarios reales.
- **Actividades clave**:
  1. Pruebas de captura de datos:
     - Realizar mediciones a diferentes distancias y en distintos entornos.
     - Evaluar la estabilidad y precisión de la transmisión de datos.
  2. Validación de datos obtenidos:

- Comparar los datos capturados con valores de referencia para asegurar su exactitud.
- Detectar y solucionar posibles anomalías.

3. Revisión del rendimiento:
   - Medir la eficiencia del sistema en condiciones de uso continuo.
   - Identificar cuellos de botella y optimizar los procesos críticos.

2.2.5. Documentación y Entrega

- **Duración**: 3 semanas.
- **Objetivo principal**: Generar una documentación completa que facilite la comprensión y uso del sistema.
- **Actividades clave**:

1. Redacción de la memoria del proyecto:
   - Detallar cada etapa del desarrollo, incluyendo logros y desafíos superados.
   - Incorporar explicaciones técnicas claras y accesibles.
2. Creación de diagramas y anexos:
   - Incluir diagramas actualizados que reflejen el estado final del sistema.
   - Elaborar manuales de usuario y guías de instalación.
3. Revisión final:
   - Validar la coherencia y calidad de la documentación.
   - Realizar ajustes finales antes de la entrega oficial.

2.3. Diagrama de Actividades

A continuación se presenta un diagrama que muestra la duración aproximada de cada etapa:

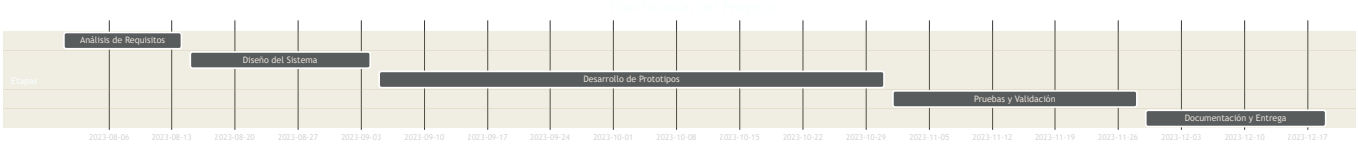

Figura 2.1: Diagrama de Gantt

2.4. Presupuesto en Horas

El tiempo invertido en cada etapa ha sido estimado de la siguiente manera:

Etapa	**Horas estimadas**
Análisis de Requisitos	40 horas
Diseño del Sistema	60 horas
Desarrollo de Prototipos	160 horas
Pruebas y Validación	80 horas
Documentación y Entrega	60 horas
Total	**400 horas**

Tabla 2.1: Estimación de horas por etapa del proyecto

Capítulo 3

Contexto Tecnológico

En este capítulo se presentan, en primer lugar, unas nociones introductorias sobre la tecnología de sensores MEMS, así como sobre Arduino y Flask, las cuales constituyen la base tecnológica de este proyecto. Se proporciona una breve descripción de las características y capacidades de los sensores MEMS utilizados para la captura de datos físicos en tiempo real, y se detallan las funcionalidades de la placa Arduino Nano IoT 33, seleccionada por su capacidad para procesar y transmitir datos de manera inalámbrica. Finalmente, se expone el rol del servidor Flask en el procesamiento y visualización de los datos, permitiendo un análisis preciso y almacenamiento efectivo para su posterior estudio.

3.1. Sensores MEMS (Microelectromechanical Systems)

La tecnología de sistemas microelectromecánicos, conocida como MEMS (Microelectromechanical Systems), fue desarrollada originalmente en los sectores automotriz y aeroespacial para sistemas de navegación y control de estabilidad. Aprovechando su capacidad para detectar movimientos finos y variaciones en aceleración y orientación, esta tecnología ha evolucionado notablemente en las últimas décadas. Con el crecimiento de los dispositivos móviles y wearables, los sensores MEMS se han convertido en una tecnología esencial no solo en la electrónica de consumo, sino también en aplicaciones relacionadas con la salud y el deporte.

3.1.1. Principios de Funcionamiento de los Sensores MEMS

Los sensores MEMS combinan microestructuras mecánicas con componentes electrónicos en chips de tamaño micrométrico. Utilizan técnicas avanzadas de micromaquinado, que permiten crear estructuras tridimensionales complejas a partir de materiales semiconductores, como el silicio. Estas estructuras son capaces de detectar cambios en magnitudes físicas, como aceleración, rotación y presión, y convertirlas en señales eléctricas. Esta funcionalidad se basa en cambios en propiedades como la capacitancia, la resistencia o incluso el magnetismo, generando una señal eléctrica que representa cuantitativamente la magnitud medida. Este tipo de tecnología es clave en aplicaciones de dispositivos wearables, teléfonos inteligentes y equipos deportivos, donde se necesita capturar datos de movimiento en tiempo real sin interferir en la actividad del usuario.

- **Acelerómetros MEMS**: Los acelerómetros MEMS están diseñados para detectar aceleración lineal en uno o varios ejes. Internamente, contienen masas suspendidas (también llamadas masas inerciales) que se desplazan ligeramente cuando el dispositivo experimenta una aceleración. Este desplazamiento provoca una variación en la capacitancia entre la masa y las placas fijas del sensor o, en algunos diseños, una variación en la resistencia. Este cambio es detectado y convertido en una señal eléctrica proporcional a la aceleración medida. En el contexto de una raqueta de bádminton, los acelerómetros permiten captar la intensidad y la dirección de los golpes, facilitando un análisis detallado de la fuerza aplicada por el jugador en cada movimiento.

- **Giroscopios MEMS**: Los giroscopios MEMS miden la velocidad angular y cambios en la orientación mediante el uso del efecto Coriolis, un fenómeno que genera una fuerza perpendicular al eje de rotación cuando el sensor se mueve en un entorno rotacional. Los giroscopios contienen osciladores internos que se desvían mínimamente ante rotaciones, permitiendo la detección precisa de cambios en orientación y velocidad angular. Esta tecnología es fundamental en deportes de raqueta, como el bádminton, ya que permite registrar la rotación de la raqueta en cada golpe y entender la dinámica del movimiento del jugador, lo cual es crítico para optimizar la técnica de golpeo y reducir el riesgo de lesiones por movimientos repetitivos.

3.1.2. Características y Ventajas de los Sensores MEMS en el Deporte

La tecnología MEMS aporta múltiples beneficios en el ámbito deportivo, destacando su precisión y capacidad de integrar varios sensores en un solo chip, lo cual permite obtener una visión detallada de los movimientos en tiempo real. A continuación, se presentan sus principales características:

- **Velocidad de Captura**: Los sensores MEMS pueden registrar movimientos a tasas de muestreo que superan los 1000 Hz, lo cual es esencial para captar y analizar movimientos rápidos en deportes de alta intensidad como el bádminton. Estas altas tasas de muestreo aseguran que incluso los detalles más sutiles de cada golpe y cambio de dirección sean registrados y analizados.

- **Alta Resolución y Precisión**: La capacidad de los MEMS para captar variaciones a nivel micrométrico permite obtener datos de aceleración y rotación con una resolución elevada, lo cual es esencial para el análisis biomecánico. Esta precisión facilita la identificación de diferencias mínimas en el rendimiento y en la técnica del jugador, aspectos difíciles de detectar a simple vista.

- **Portabilidad y Ligereza**: Con tamaños reducidos que pueden ser de apenas unos pocos milímetros, los sensores MEMS pueden integrarse en dispositivos portátiles y ligeros, como una raqueta de bádminton, sin impactar en el peso o en el rendimiento del jugador. Esto permite al atleta jugar sin interferencias, mientras los sensores registran la información detallada de cada movimiento.

- **Bajo Consumo de Energía**: Una de las grandes ventajas de los MEMS es su eficiencia energética, lo cual los hace ideales para dispositivos portátiles y wearables que necesitan autonomía para funcionar por periodos prolongados. Esto es particularmente útil en sesiones de entrenamiento

o partidos de larga duración, donde es importante que el dispositivo capture datos continuos sin necesidad de recargar.

- **Compatibilidad con Tecnologías Inalámbricas**: Los sensores MEMS están diseñados para ser compatibles con sistemas inalámbricos como Bluetooth o Wi-Fi, permitiendo la transmisión en tiempo real de los datos capturados. Esto posibilita un análisis remoto y monitoreo continuo del rendimiento del atleta, facilitando la toma de decisiones en tiempo real. En el ámbito del bádminton, los datos transmitidos en directo permiten que entrenadores y analistas deportivos ajusten estrategias de entrenamiento y realicen correcciones de técnica basadas en información detallada de los movimientos.

- **Integración en Dispositivos Móviles y *Wearables***: La versatilidad de los sensores MEMS permite su incorporación en dispositivos móviles y *wearables*, lo cual amplía las aplicaciones de su uso en el deporte. Desde relojes inteligentes hasta aplicaciones de monitoreo en tiempo real, los sensores MEMS proporcionan datos accesibles y personalizables, adaptándose a las necesidades individuales del atleta y ofreciendo una mayor conectividad en el ecosistema deportivo.

3.2. Plataforma Arduino - Atmel y Placas con Sensores Arduino

La plataforma Arduino, basada en microcontroladores de la familia Atmel AVR, se ha convertido en una de las herramientas de referencia para proyectos de electrónica y sistemas embebidos. Sus placas ofrecen una infraestructura sencilla para desarrollar y prototipar aplicaciones interactivas gracias a su facilidad de programación, su entorno de desarrollo accesible y su capacidad de integración con diversos sensores y módulos de comunicación.

3.2.1. Características de la Plataforma Arduino

Arduino permite una interfaz amigable y accesible para programar microcontroladores que pueden gestionar y procesar entradas de sensores. Uno de sus principales microcontroladores es el ATmega328 (utilizado en Arduino Uno), mientras que para aplicaciones con necesidades avanzadas de conectividad, se emplean microcontroladores de la serie SAMD21 en la línea Arduino Nano, ofreciendo capacidades mejoradas de comunicación inalámbrica.

Ventajas de la plataforma Arduino:

- **Facilidad de uso**: Su lenguaje de programación es una versión simplificada de C++, con una amplia biblioteca de ejemplos y documentación.

- **Versatilidad**: Compatible con múltiples sensores y módulos de comunicación, incluyendo tecnologías de bajo consumo y comunicaciones inalámbricas.

- **Ecosistema abierto**: Al ser de código abierto, la plataforma cuenta con una comunidad global que facilita el acceso a módulos, sensores y mejoras constantes en el hardware y software.

3.2.2. Placas Arduino Nano con Sensores: Capacidades, Tamaño y Consumo

Las placas de la línea Arduino Nano han sido diseñadas con un tamaño compacto que permite la integración en proyectos donde el espacio es limitado, como en el caso de una raqueta de bádminton. Dentro de esta línea, la Arduino Nano 33 IoT y la Arduino Nano 33 BLE se destacan por sus capacidades de conexión inalámbrica y compatibilidad con sensores.

Arduino Nano 33 IoT

- **Microcontrolador**: SAMD21 Cortex-M0+ de 32 bits, con capacidad de procesamiento adecuada para proyectos con sensores en tiempo real.
- **Conectividad**: Incluye un chip Wi-Fi y Bluetooth de la serie NINA-W102, que permite transmitir datos en tiempo real de forma inalámbrica, una ventaja significativa para monitorear datos sin cables en deportes.
- **Tamaño**: 45 x 18 mm, lo cual es ideal para su integración en la empuñadura de una raqueta sin afectar el peso ni el manejo.
- **Consumo energético**: Bajo consumo en modo de espera, con un promedio de 19 mA en operación completa y 6-7 mA en modo de ahorro de energía. Estos valores permiten sesiones de uso prolongado con una batería pequeña.

Arduino Nano 33 BLE

- **Microcontrolador**: nRF52840 con procesador Cortex-M4F, adecuado para aplicaciones que requieren procesamiento local de datos antes de enviarlos, gracias a su capacidad de procesamiento superior.
- **Conectividad**: Soporte exclusivo para Bluetooth 5.0, que permite tasas de transferencia elevadas y un rango ampliado, ideal para analizar movimientos en una cancha de bádminton.
- **Tamaño**: Similar al Arduino Nano 33 IoT, conservando la ventaja de portabilidad.
- **Consumo energético**: 15-20 mA en transmisión activa, con modos de bajo consumo que reducen el uso energético, extendiendo la autonomía en aplicaciones portátiles.

3.2.3. Evaluación de Adecuación para el Proyecto

Dado que el objetivo es capturar datos en tiempo real sobre la dinámica de una raqueta de bádminton, tanto el tamaño como el consumo energético de las placas Arduino Nano 33 IoT y BLE las hacen aptas para el problema planteado. Sus capacidades inalámbricas permiten la captura y transmisión de datos sin cables, lo que facilita el monitoreo de variables como aceleración y velocidad angular durante el juego.

Además, las dimensiones y peso de estas placas son lo suficientemente pequeñas como para integrarse sin inconvenientes en el mango de una raqueta, y el bajo consumo asegura que puedan funcionar durante una sesión completa de entrenamiento sin requerir recarga.

Un factor clave en la decisión de emplear Wi-Fi en lugar de Bluetooth Low Energy (BLE) fue la diferencia en los enfoques de programación. Mientras que BLE se apoya en un modelo basado en servicios, que proporciona una mayor abstracción para el manejo de datos, esta misma característica introduce una complejidad adicional tanto en su programación como en las pruebas. BLE, aunque soportado por bibliotecas específicas, requiere un entendimiento profundo de sus conceptos de servicios y características para diseñar una comunicación eficiente. En contraste, Wi-Fi utiliza la pila IP, ampliamente conocida y documentada, lo que facilita tanto su implementación como la depuración de problemas. Esto hace que Wi-Fi sea más adecuado para un proyecto que prioriza la simplicidad de desarrollo y el manejo eficiente de datos en tiempo real.

Conclusión: Aunque ambas placas cumplen con los requisitos de tamaño, consumo y conectividad, la elección de la Arduino Nano 33 IoT y su uso de Wi-Fi se justifica por la simplicidad en el desarrollo y las ventajas en la transmisión de datos en tiempo real. Esto asegura una solución robusta y eficiente para el análisis de movimientos en el deporte.

3.3. Servidores Flask y Comunicación en Tiempo Real

En el desarrollo de aplicaciones modernas, especialmente en el campo de la captura y análisis de datos en tiempo real, la elección del framework y el lenguaje de programación adecuado es crucial. En este contexto, Flask se presenta como una opción ideal para gestionar la comunicación entre la placa Arduino Nano IoT 33 y el sistema que recibe, procesa y visualiza los datos. Flask es un microframework para Python que se utiliza comúnmente para desarrollar aplicaciones web ligeras y eficientes. En este proyecto, se emplea para crear un servidor que pueda recibir los datos provenientes de la raqueta de bádminton y gestionarlos de forma efectiva.

3.3.1. ¿Por qué Flask y Python?

Flask se destaca por su simplicidad y flexibilidad. A diferencia de otros frameworks más complejos, Flask no impone una estructura rígida, lo que permite al desarrollador tener un control completo sobre cómo se organiza la aplicación. Esta característica lo convierte en una excelente opción para proyectos que requieren una implementación rápida y personalizada, como el análisis de datos en tiempo real.

Python, por su parte, es uno de los lenguajes de programación más populares debido a su facilidad de aprendizaje, sintaxis limpia y un ecosistema muy robusto de bibliotecas, como **NumPy**, **Pandas** y **Matplotlib**, que facilitan el procesamiento y la visualización de datos. Además, Python es ampliamente utilizado en el campo de la ciencia de datos y el análisis en tiempo real, lo que lo convierte en la opción natural para el desarrollo de proyectos que implican la manipulación de grandes volúmenes de información, como es el caso en este proyecto de sensores MEMS.

3.3.2. Comparativa: Flask (Python) vs Node.js

Al considerar tecnologías para el desarrollo de un servidor que maneje la comunicación en tiempo real, Node.js es otro marco comúnmente utilizado, especialmente en aplicaciones web y de IoT debido a su

eficiencia y capacidad para manejar múltiples conexiones concurrentes de forma no bloqueante.

Ventajas de Flask (Python):

- **Simplicidad y Flexibilidad**: Flask es minimalista y permite construir aplicaciones a medida, sin imposiciones. Esto es útil cuando se necesita personalizar un servidor para procesar datos en tiempo real de manera precisa.
- **Bibliotecas de Ciencia de Datos**: Python tiene un ecosistema muy fuerte en ciencia de datos, lo cual es esencial para proyectos que requieren procesar, analizar y visualizar grandes volúmenes de datos en tiempo real.
- **Fácil integración con sensores y plataformas de IoT**: Flask es compatible con una amplia variedad de protocolos de comunicación como HTTP y MQTT, lo que facilita la integración con sensores como los MEMS.

Ventajas de Node.js:

- **Alta concurrencia**: Node.js utiliza un modelo basado en eventos, lo que le permite manejar un gran número de conexiones simultáneas con una menor sobrecarga de recursos, lo cual es útil en aplicaciones de IoT con muchos dispositivos.
- **Desarrollo en JavaScript**: Al usar JavaScript tanto en el frontend como en el backend, Node.js puede ofrecer una experiencia más fluida para los desarrolladores con experiencia en este lenguaje.
- **Rendimiento en tiempo real**: Node.js es particularmente eficiente en tareas que implican comunicación en tiempo real, como la mensajería instantánea o la transmisión de datos en vivo.

Desventajas de Flask (Python):

- **Concurrencia limitada**: Python no está tan optimizado para manejar tareas de alta concurrencia como Node.js, aunque esto se puede mitigar con el uso de bibliotecas y arquitecturas como Celery o Threading.
- **Menos adecuado para aplicaciones con gran cantidad de usuarios simultáneos**: Flask es adecuado para aplicaciones que no necesiten un alto nivel de concurrencia, pero podría no ser tan eficiente como Node.js en aplicaciones de muy alta escala.

Desventajas de Node.js:

- **Curva de aprendizaje para desarrolladores no familiarizados con JavaScript**: Aunque JavaScript es muy popular, algunos desarrolladores pueden encontrar más sencillo trabajar con Python debido a su sintaxis más legible.
- **Limitado para aplicaciones que requieren procesamiento intensivo de datos**: Aunque Node.js es eficiente para manejar conexiones, no es tan fuerte en tareas de análisis de datos o procesamiento pesado como Python.

La elección de **Flask** y **Python** para este proyecto se justifica por la simplicidad del microframework, su compatibilidad con tecnologías de IoT, y la robustez del ecosistema de Python para la manipulación y análisis de datos. Si bien **Node.js** también es una opción válida, especialmente para aplicaciones de alta concurrencia, **Flask** ofrece un entorno más adecuado para el procesamiento y almacenamiento de datos científicos, lo cual es crucial en el análisis de datos de sensores MEMS en tiempo real. Además, esta decisión fue tomada bajo la orientación de mi tutor, quien me recomendó utilizar **Flask** por considerar que, además de ser una opción interesante para el proyecto, me permitiría explorar una tecnología nueva y valiosa en el ámbito de la programación web.

3.3.3. Funciones Clave del Servidor Flask en el Proyecto

En el contexto de este proyecto, el servidor Flask desempeña varias funciones críticas para la gestión y análisis de los datos capturados por los sensores de la raqueta. Entre sus principales responsabilidades se incluyen:

- **Recepción de datos en tiempo real**: El servidor Flask recibe los datos enviados por la placa Arduino Nano IoT 33, que está conectada a los sensores MEMS. Esta comunicación puede ser continua o por intervalos configurables, lo que permite adaptar la toma de datos a las necesidades del usuario.
- **Procesamiento y almacenamiento de datos**: Una vez que el servidor recibe los datos, los procesa y los almacena para su posterior análisis. Esto incluye la validación de la información y el guardado en bases de datos o archivos, como archivos CSV, para permitir un análisis detallado.
- **Interfaz de comunicación**: Flask proporciona una interfaz accesible para interactuar con el sistema. Esto permite al usuario iniciar o detener la toma de datos, ajustar configuraciones, verificar el estado de los sensores, o incluso visualizar los resultados en tiempo real.
- **Protocolos de comunicación**: El servidor utiliza protocolos de comunicación como HTTP o MQTT. HTTP es un protocolo estándar para aplicaciones web, adecuado para la recepción de solicitudes y la transmisión de datos.
- **Visualización de datos**: El servidor Flask también puede integrarse con herramientas de visualización de datos, como Chart.js o Plotly, que permiten presentar los datos de manera gráfica en tiempo real. Esto es útil para los usuarios que necesitan monitorear el rendimiento de los sensores o los movimientos de la raqueta durante el juego.

Capítulo 4

Análisis y Diseño

4.1. Requisitos

Para llevar a cabo el diseño de un sistema de adquisición de datos eficiente, es fundamental definir claramente los requisitos que este debe cumplir. Estos requisitos se dividen en dos grandes categorías: requisitos funcionales y requisitos no funcionales. Los primeros describen las funciones y capacidades específicas que el sistema debe ofrecer, como la captura, procesamiento y visualización de datos. Por otro lado, los requisitos no funcionales se refieren a aspectos de calidad del sistema, como el rendimiento, la usabilidad, la fiabilidad, la seguridad y la documentación necesaria para asegurar que el sistema sea útil y sostenible a largo plazo.

Además de estos requisitos, el sistema debe respetar también restricciones de diseño impuestas en la arquitectura inicial del proyecto, como la selección de la plataforma hardware, el stack de programación utilizado y la interfaz de usuario. Estas restricciones forman parte de las decisiones clave tomadas en etapas previas para garantizar la coherencia y viabilidad del desarrollo. A continuación, se detallan ambos tipos de requisitos.

4.1.1. Requisitos funcionales

1. **Captura de datos:**

   - **Medición en tiempo real:** El sistema debe ser capaz de capturar datos de aceleración y velocidad angular en tiempo real desde los sensores MEMS integrados en la raqueta de bádminton. Los datos deben incluir parámetros clave como la aceleración lineal (en los ejes X, Y y Z), velocidad angular, y la orientación de la raqueta durante el juego.
   - **Modos de captura:**
     - **Captura continua:** El sistema debe permitir la captura de datos de manera continua durante el tiempo que el usuario lo desee.
     - **Captura por intervalo de tiempo:** El sistema debe ser capaz de capturar datos durante un intervalo de tiempo específico, configurado por el usuario.

- **Transmisión de datos:** Los datos capturados por los sensores deben ser transmitidos al servidor Flask para su procesamiento. La transmisión debe realizarse de manera eficiente y sin pérdidas de datos.

2. **Interfaz de usuario:**

   - **Visualización en el momento:** El sistema debe proporcionar una interfaz gráfica para la visualización de datos en momento. Esto incluye gráficos que muestran la aceleración, velocidad angular y orientación de la raqueta.
   - **Controles de interacción:** La interfaz debe permitir al usuario iniciar, detener y configurar la toma de datos. También debe ofrecer opciones para visualizar los datos en diferentes formatos y ajustar los parámetros de captura.

3. **Procesamiento de datos:**

   - **Validación de datos:** El sistema debe incluir mecanismos para validar la precisión de los datos capturados. En caso de detectar datos incorrectos o inconsistentes, el sistema debe informar al usuario y permitir realizar ajustes.

4. **Almacenamiento de datos:**

   - **Archivos de datos:** Los datos válidos deben ser almacenados en archivos o bases de datos para su análisis posterior. El sistema debe gestionar el almacenamiento de manera eficiente, asegurando la integridad y disponibilidad de los datos.

5. **Configuración y ajustes:**

   - **Configuración del sistema:** El sistema debe permitir al usuario ajustar los parámetros de captura de datos, como la frecuencia de muestreo y el intervalo de tiempo para la captura. Estos ajustes deben ser accesibles a través de la interfaz de usuario.

4.1.2. Requisitos no funcionales

1. **Desempeño:**

   - **Tiempo de respuesta:** El sistema debe proporcionar datos con un tiempo de respuesta mínimo, asegurando que la captura y visualización en el momento se realicen sin retrasos significativos. Esto es crucial para una evaluación precisa del rendimiento durante el juego.

2. **Usabilidad:**

   - **Interfaz intuitiva:** La interfaz de usuario debe ser fácil de usar y comprender, incluso para personas sin experiencia técnica. Debe incluir instrucciones claras y opciones de ayuda para facilitar el uso del sistema.

- **Accesibilidad:** La interfaz debe ser accesible desde diferentes dispositivos y plataformas, permitiendo a los usuarios interactuar con el sistema desde ordenadores, tabletas o teléfonos móviles.

3. **Fiabilidad:**
   - **Exactitud de datos:** Los sensores y el sistema deben asegurar la precisión en la captura y procesamiento de datos. La calibración de los sensores debe ser revisada periódicamente para mantener la exactitud de las mediciones.
   - **Robustez:** El sistema debe ser robusto y capaz de operar en condiciones variables sin fallos. Debe incluir mecanismos para detectar y manejar errores de manera efectiva.
   - **Mantenimiento:** El sistema debe ser fácil de mantener, con procedimientos claros para realizar ajustes, actualizaciones y correcciones de errores. Debe incluir documentación y soporte para el mantenimiento regular.
4. **Seguridad:**
   - **Proteción de datos:** Los datos recogidos deben ser protegidos contra accesos no autorizados. El sistema debe incluir medidas de seguridad para garantizar la privacidad y la integridad de los datos almacenados.
5. **Documentación:**
   - **Manual de usuario:** Debe incluirse un manual de usuario detallado que explique cómo utilizar el sistema, configurar los parámetros de captura, y visualizar y analizar los datos.
   - **Documentación técnica:** La documentación técnica debe proporcionar información detallada sobre la arquitectura del sistema, el diseño de los componentes, y los procedimientos de instalación y mantenimiento.

4.2. Modelo de dominio y relación con el diseño

El modelo de dominio es una representación conceptual que define los principales elementos del sistema y sus relaciones. Este modelo proporciona una visión clara de los datos y su organización, sirviendo como base para el diseño detallado del sistema.

En esta sección se presentan dos diagramas:

1. **Diagrama de dominio en análisis**, que refleja los conceptos clave del sistema sin entrar en detalles de implementación.
2. **Diagrama de dominio en diseño**, que adapta el modelo anterior para reflejar las estructuras que se utilizarán en la implementación del sistema.

4.2.1. Diagrama de dominio en análisis

El diagrama de dominio en análisis se basa en los conceptos fundamentales extraídos de los casos de uso del sistema. Este diagrama representa los elementos principales sin entrar en detalles de implementación.

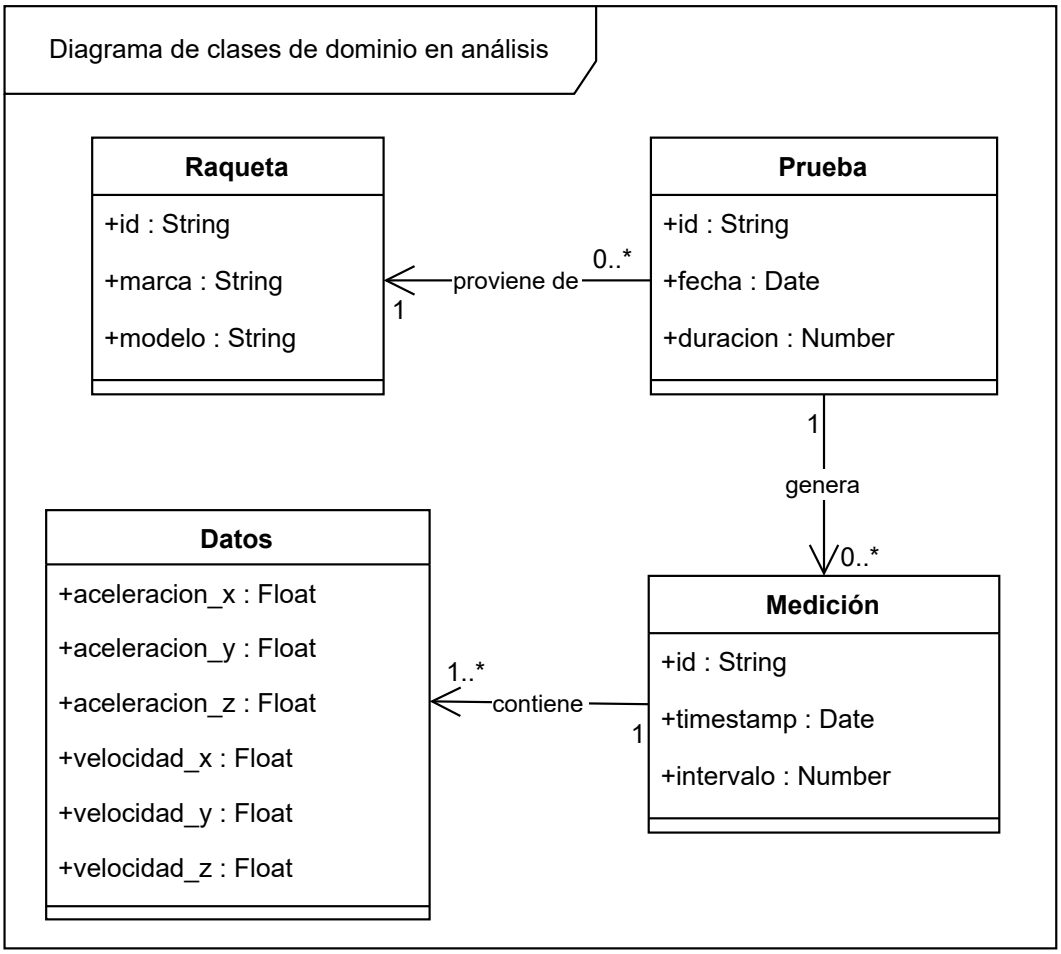

Figura 4.1: Diagrama de clases de dominio

Las entidades clave del sistema son:

- **Raqueta**: Representa la raqueta sensorizada utilizada en la captura de datos.
- **Prueba**: Sesión en la que se registran datos capturados durante un periodo de tiempo.
- **Serie**: Conjunto de datos registrados en un intervalo de tiempo dentro de una prueba.
- **Datos**: Representa los valores obtenidos de los sensores, incluyendo aceleración y velocidad en los ejes X, Y y Z.

Las relaciones principales entre estas entidades son:

- Una **Prueba** genera múltiples Series de datos.

- Cada **Serie** contiene múltiples Datos capturados en intervalos de tiempo específicos.
- Cada **Prueba** está asociada a una **Raqueta**, identificando el equipo utilizado.

4.2.2. Transición del análisis al diseño

A partir del modelo de dominio en análisis, es posible refinar la estructura para alinear los elementos del sistema con los casos de uso y la arquitectura de implementación. Esto implica:

1. Asignar nombres consistentes con los diagramas de casos de uso en diseño.
2. Diferenciar las entidades de dominio de los componentes de software.
3. Especificar las interacciones necesarias para la implementación.

4.2.3. Diagrama de clases de dominio en diseño

El diagrama de dominio en diseño es una versión refinada del modelo de análisis, ajustada para reflejar la implementación real del sistema.

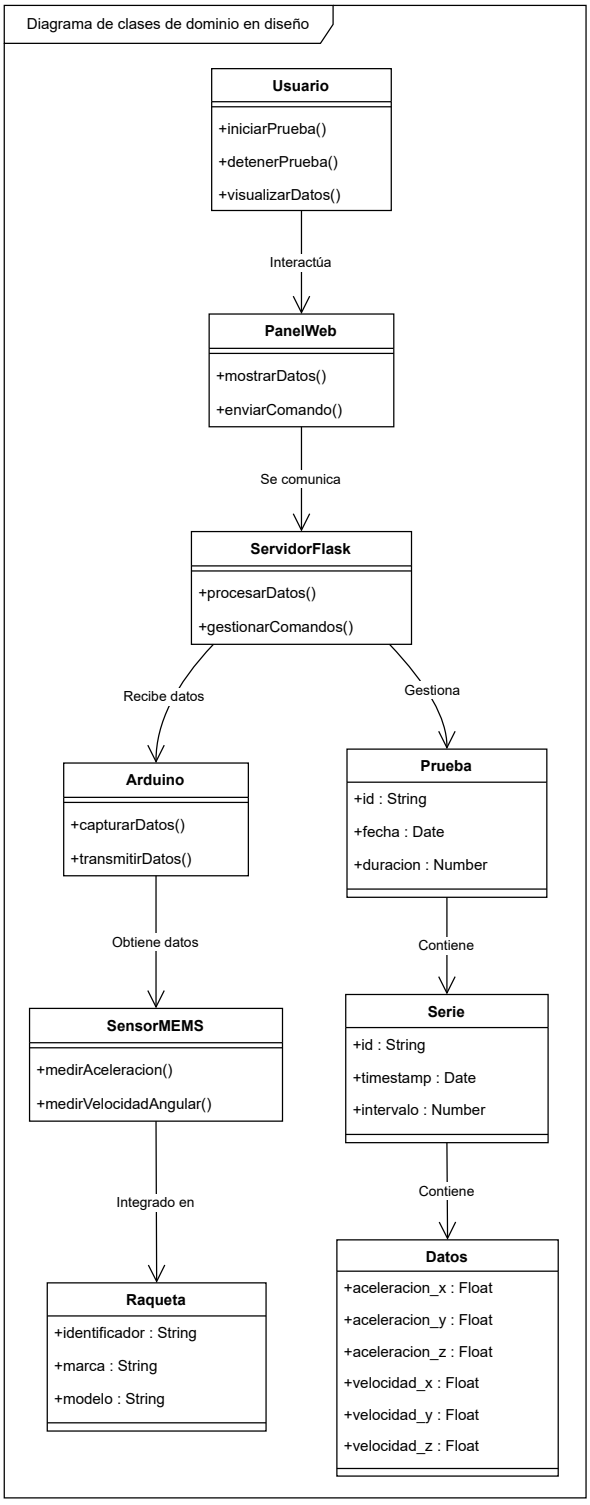

Figura 4.2: Diagrama de clases de dominio en diseño

Las principales modificaciones incluyen:

- Incorporación de nuevas clases: Se añaden entidades clave como el Usuario, el Panel Web, el Servidor Flask.
- Estructuración de la comunicación: Se reflejan las interacciones entre los sensores MEMS, el servidor y la interfaz web.
- Relaciones ajustadas: Se detalla cómo el usuario interactúa con la aplicación y cómo se gestionan las pruebas y series de datos en el sistema.

4.3. Casos de uso

En este apartado se presentan los posibles casos de uso del sistema desarrollado para la adquisición y monitorización de datos provenientes de sensores MEMS instalados en una raqueta de bádminton.

El objetivo del sistema es permitir la captura, visualización y almacenamiento de datos relacionados con las aceleraciones y velocidades angulares de la raqueta durante su uso. La interacción con el sistema se realiza a través de un panel web, desde el cual el usuario puede iniciar o detener la adquisición de datos, realizar pruebas de sensores, así como ajustar ciertos parámetros como el tiempo de muestreo.

Cada caso de uso refleja una posible interacción del usuario con el sistema, describiendo tanto las funcionalidades disponibles como los flujos de trabajo principales. Estos casos de uso permiten visualizar cómo el usuario puede controlar el sistema, verificar su correcto funcionamiento, y asegurar que los datos obtenidos son válidos para su análisis posterior. Además, se destacan las precondiciones necesarias para ejecutar cada funcionalidad y las postcondiciones que aseguran la correcta operación del sistema.

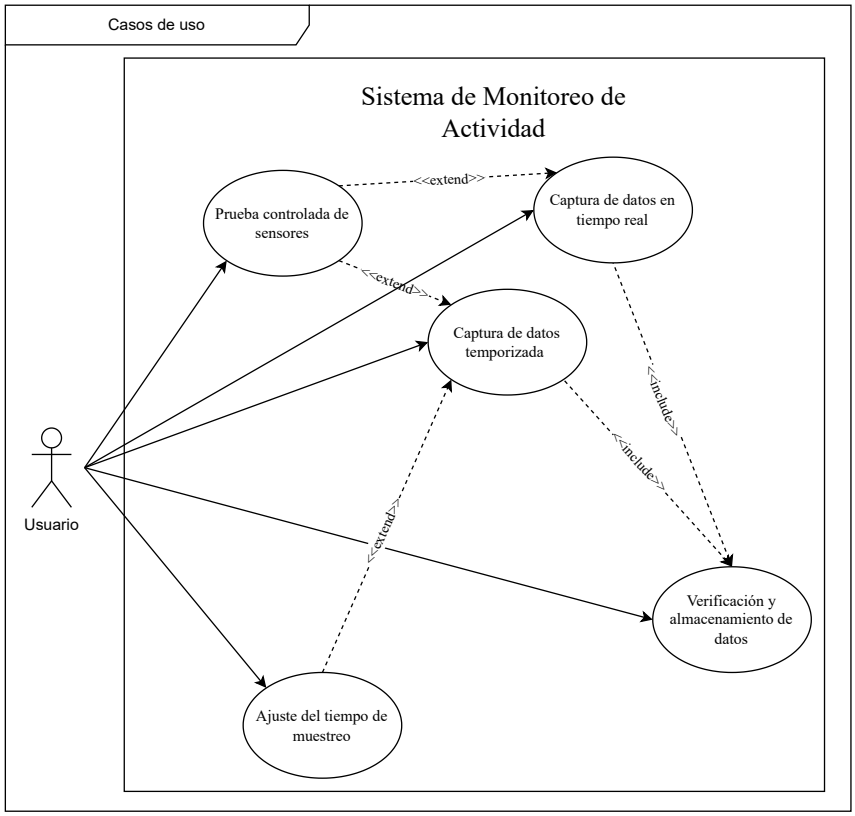

Figura 4.3: Casos de uso

4.3.1. Captura de datos en tiempo real

CU-1	Captura de datos en tiempo real
Actor	Usuario
Descripción	El usuario inicia la captura de datos en tiempo real a través del panel web.
Precondición	El sistema está operativo y conectado a los sensores MEMS en la raqueta.
Secuencia	El usuario accede al panel web y hace clic en el botón “COMENZAR”. El servidor Flask envía la señal de inicio de captura continua al Arduino. Arduino inicia la captura continua de los datos de aceleración y velocidad angular. Los datos son transmitidos al servidor Flask en tiempo real para su procesamiento. El usuario puede detener la captura en cualquier momento haciendo clic en “PARAR”.
Postcondiciones	Los datos capturados se almacenan para un análisis posterior si el usuario lo solicita.

Tabla 4.1: Caso de uso 1 - Captura de datos en tiempo real

En este caso de uso, el sistema permite la captura continua de datos provenientes de los sensores MEMS integrados en la raqueta de bádminton. El usuario accede al panel web, una interfaz gráfica intuitiva, y selecciona la opción de comenzar la captura de datos en tiempo real. Durante este proceso, el sistema adquiere mediciones de aceleración y velocidad angular en los tres ejes (X, Y, Z) de manera continua. Los datos son enviados al servidor Flask a través de una conexión inalámbrica para su procesamiento en tiempo real.

Una vez procesados, los datos son mostrados en gráficos interactivos disponibles en el panel web, lo que permite al usuario observar el comportamiento dinámico de la raqueta durante su uso. Este modo de operación es particularmente útil para analizar movimientos prolongados, ya que el sistema no impone un límite de tiempo mientras la captura esté activa. En cualquier momento, el usuario tiene la posibilidad de detener el proceso presionando el botón correspondiente en la interfaz. Adicionalmente, si el usuario lo solicita, los datos obtenidos durante la sesión pueden ser almacenados para su análisis posterior en formato CSV. Este flujo asegura que el sistema ofrezca una experiencia de uso continua y accesible, facilitando tanto la visualización como la gestión de los datos obtenidos.

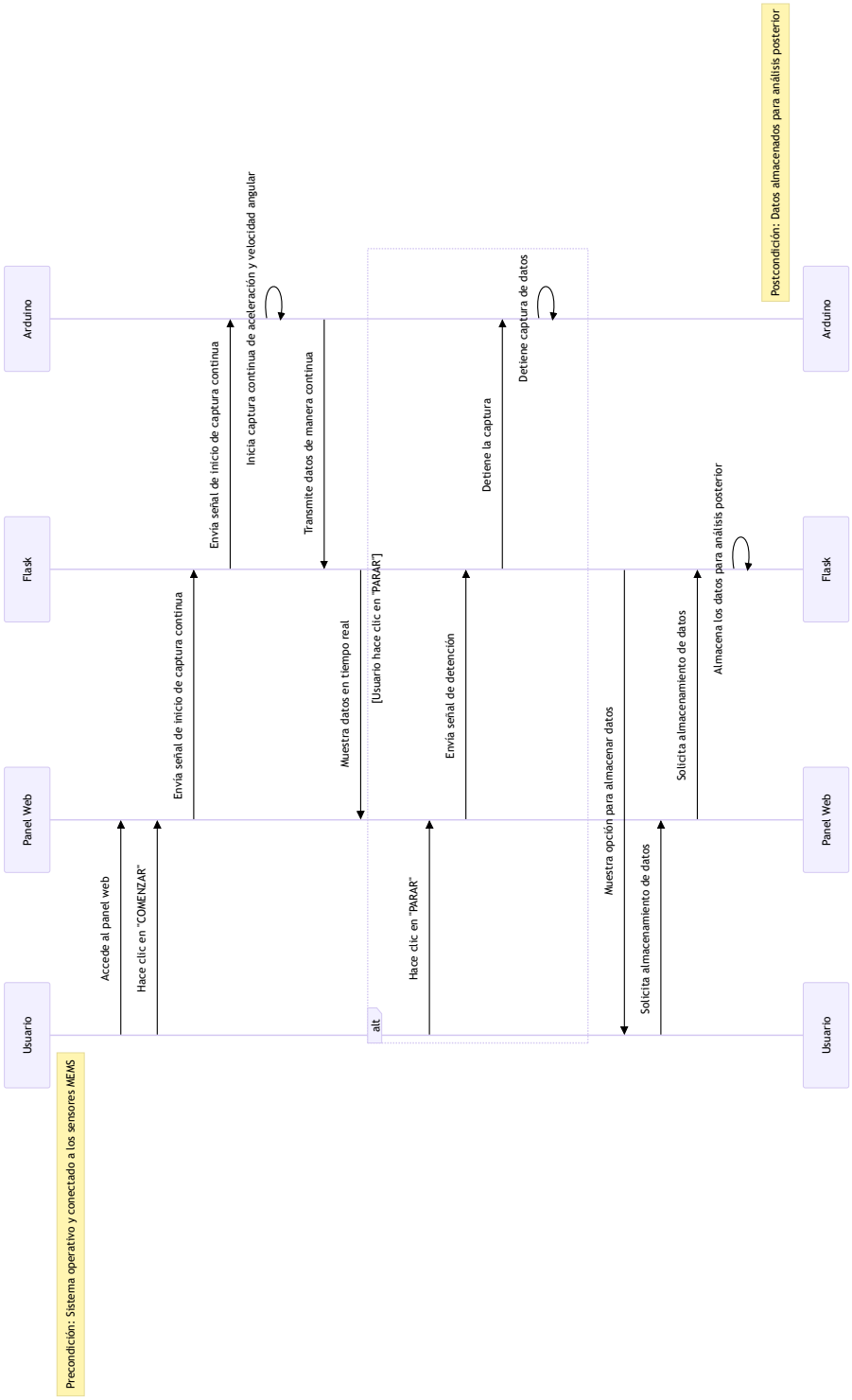

Figura 4.4: Diagrama de secuencia del caso de uso CU-1

4.3.2. Captura de datos temporizada

CU-2	Captura de datos temporizada
Actor	Usuario
Descripción	El usuario realiza una captura de datos durante un tiempo determinado.
Precondición	El sistema está operativo y conectado a los sensores MEMS en la raqueta.
Secuencia	El usuario accede al panel web y selecciona el número de segundos en el campo “Nº seg”. El usuario hace clic en el botón “COMENZAR TEMPORIZADO”. El servidor Flask envía la señal de inicio de captura temporizada de datos al Arduino. Arduino inicia la captura de datos durante el tiempo especificado. Los datos son transmitidos al servidor Flask para su procesamiento en tiempo real. El panel web muestra un temporizador en la pantalla. Al finalizar el tiempo, la captura se detiene automáticamente.
Postcondiciones	Los datos se almacenan para un análisis posterior si se solicita.

Tabla 4.2: Caso de uso 2 - Captura de datos temporizada

El sistema también ofrece la posibilidad de realizar capturas de datos durante un intervalo de tiempo predefinido. Esta funcionalidad se inicia cuando el usuario selecciona el número de segundos deseado en el campo indicado del panel web y presiona el botón correspondiente para comenzar la captura temporizada. Una vez activada, el sistema adquiere datos durante el tiempo especificado, transmitiéndolos al servidor Flask para su procesamiento y visualización en tiempo real.

Mientras la captura está activa, un temporizador es mostrado en la pantalla, indicando el tiempo restante. Esto permite al usuario seguir de manera precisa el progreso de la medición. Al concluir el intervalo establecido, el sistema detiene automáticamente la captura y procesa los datos recopilados. Estos datos, al igual que en la captura continua, pueden ser almacenados en un archivo CSV para su uso posterior si el usuario así lo decide. Este caso de uso es especialmente útil en escenarios donde se requiere analizar una actividad específica durante un tiempo limitado, garantizando así un control riguroso del proceso de adquisición.

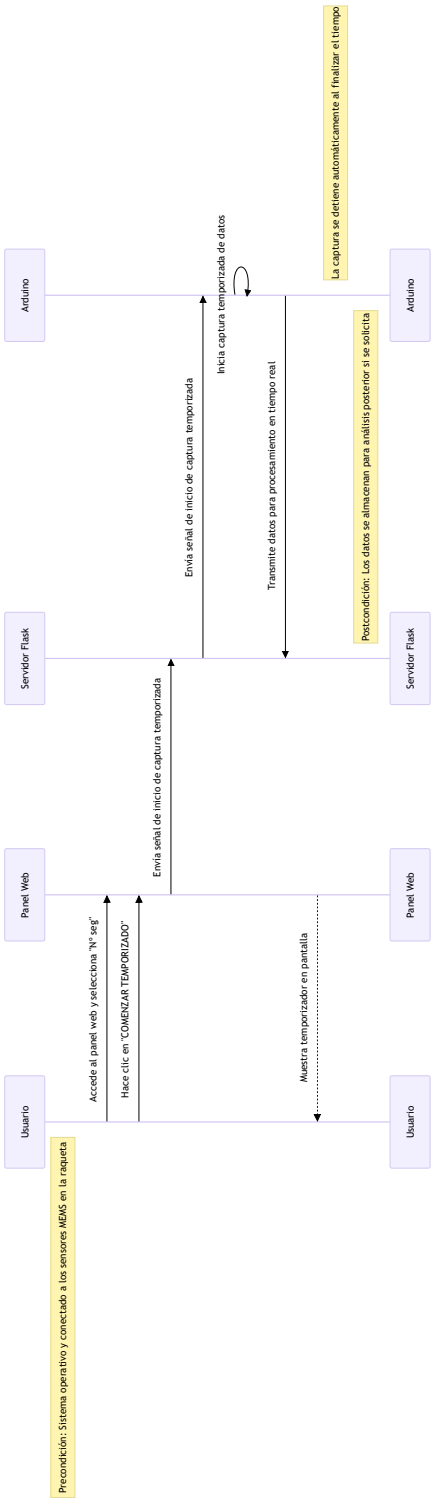

Figura 4.5: Diagrama de secuencia del caso de uso CU-2

4.3.3. Prueba controlada de sensores

CU-3	Prueba controlada de sensores
Actor	Usuario
Descripción	El usuario realiza una prueba de 10 segundos para verificar el correcto funcionamiento de los sensores.
Precondición	El sistema está operativo y conectado a los sensores MEMS en la raqueta.
Secuencia	El usuario accede al panel web y hace clic en “TESTEAR SENSORES”. El servidor Flask envía la señal de inicio de testeo de sensores al Arduino. Arduino inicia el testeo de sensores durante 10 segundos. Los datos son procesados y se muestran en los gráficos al acabar. El usuario verifica que los sensores funcionan correctamente.

Tabla 4.3: Caso de uso 3 - Prueba controlada de sensores

Con el objetivo de verificar el correcto funcionamiento de los sensores MEMS, el sistema incluye una prueba controlada con una duración fija de 10 segundos. Esta prueba se inicia desde el panel web mediante la selección de la opción ”Testear Sensores”. Una vez activada, el sistema realiza una captura de datos durante el tiempo indicado, garantizando que todas las mediciones necesarias sean tomadas.

Al finalizar la prueba, los datos capturados son procesados y visualizados mediante gráficos en el panel web, proporcionando al usuario una representación clara del rendimiento de los sensores. Esta información es reportada al usuario, quien puede tomar decisiones respecto a posibles ajustes o mantenimiento. Este caso de uso es esencial para asegurar la fiabilidad del sistema antes de su uso intensivo en análisis biomecánicos o sesiones de entrenamiento.

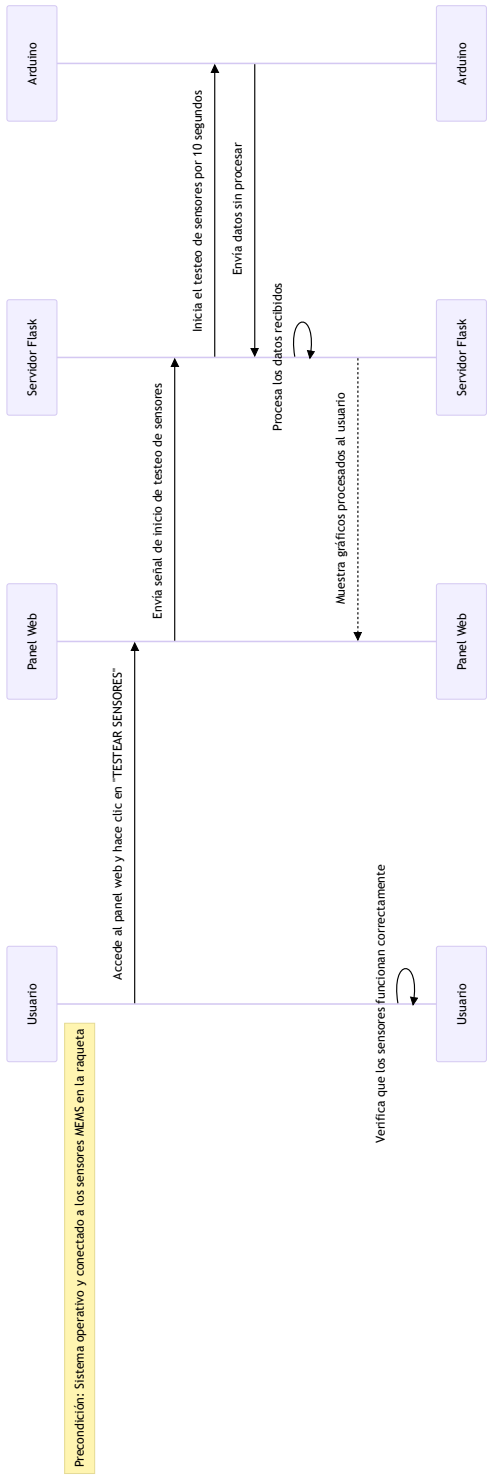

Figura 4.6: Diagrama de secuencia del caso de uso CU-3

4.3.4. Verificación y almacenamiento de datos

CU-4	Verificación y almacenamiento de datos
Actor	Usuario
Descripción	El usuario verifica que los datos capturados son correctos y puede guardarlos.
Precondición	El sistema está operativo y conectado a los sensores MEMS en la raqueta.
Secuencia	El usuario accede al panel web y hace clic en “COMPROBAR DATOS”. Los datos capturados se procesan y se muestran en gráficos. Si los datos son correctos, el usuario puede introducir un nombre de archivo y hacer clic en “GUARDAR”para almacenarlos.
Postcondiciones	Los datos se guardan en un archivo CSV en el servidor o en el dispositivo del usuario.

Tabla 4.4: Caso de Uso 4 - Verificación y almacenamiento de datos

La verificación de datos permite al usuario evaluar la calidad y precisión de las mediciones capturadas. Una vez finalizada una sesión de captura, el usuario puede acceder a esta funcionalidad desde el panel web, donde los datos se procesan y presentan en gráficos interactivos. Estos gráficos proporcionan una representación visual detallada que permite identificar patrones o anomalías en las mediciones.

Si los datos se consideran válidos, el usuario tiene la opción de guardarlos en un archivo CSV introduciendo un nombre personalizado en el campo correspondiente. Este archivo es almacenado localmente en el servidor o descargado al dispositivo del usuario para su análisis posterior. Esta funcionalidad asegura la integridad y accesibilidad de los datos, facilitando su uso en estudios comparativos o evaluaciones futuras.

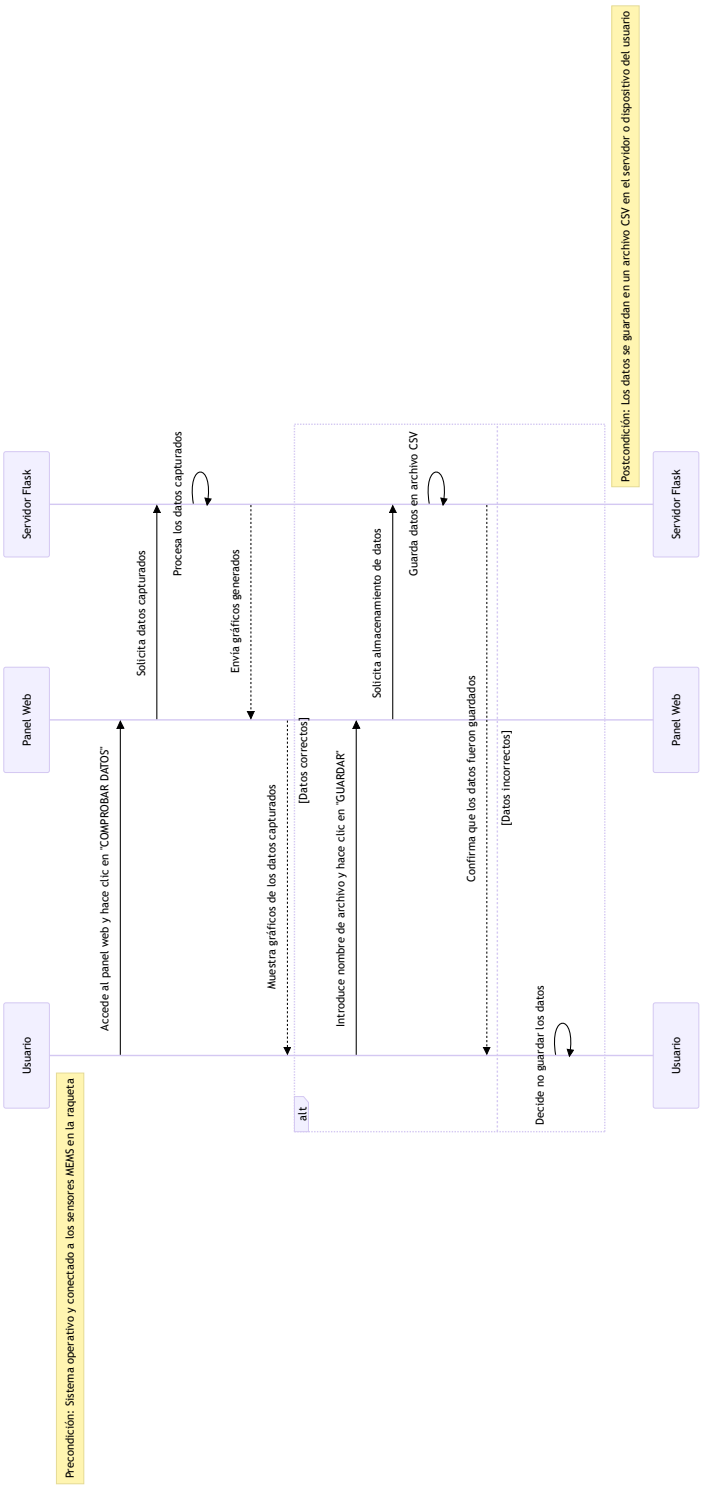

Figura 4.7: Diagrama de secuencia del caso de uso CU-4

4.3.5. Ajuste del tiempo de muestreo

CU-5	Ajuste del tiempo de muestreo
Actor	Usuario
Descripción	El usuario ajusta el tiempo de muestreo de los datos desde el panel web.
Precondición	El sistema está operativo y conectado a los sensores MEMS en la raqueta.
Secuencia	El usuario selecciona el número de segundos deseados en el campo “Nº seg”. El usuario inicia la captura temporizada con el nuevo tiempo de muestreo. El servidor Flask envía la señal de inicio de captura temporizada de datos al Arduino. Arduino inicia la captura de datos según el tiempo de muestreo configurado.
Postcondiciones	Los datos se capturan utilizando el tiempo de muestreo ajustado.

Tabla 4.5: Caso de uso 5 - Ajuste del tiempo de muestreo

Para adaptarse a diferentes escenarios de análisis, el sistema permite ajustar el tiempo de muestreo de las capturas. Esta configuración se realiza a través del panel web, donde el usuario selecciona el número de segundos deseado en el campo correspondiente. Una vez ajustado, el tiempo de muestreo se aplica inmediatamente en la siguiente captura temporizada.

Al finalizar la captura configurada con el nuevo tiempo de muestreo, los datos obtenidos son almacenados o procesados conforme a los parámetros establecidos, garantizando flexibilidad y personalización en el uso del sistema.

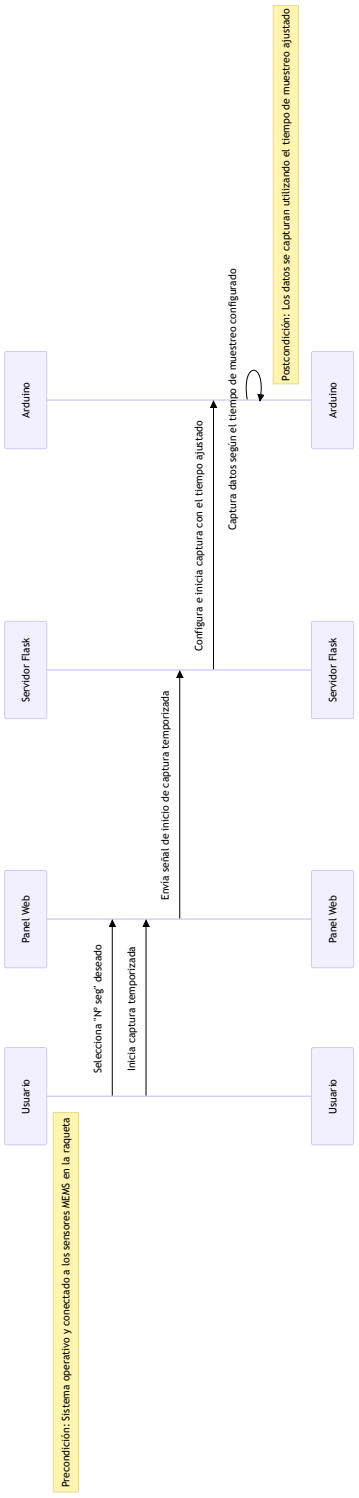

Figura 4.8: Diagrama de secuencia del caso de uso CU-5

4.4. Diagrama de arquitectura

La arquitectura del sistema se define para mostrar cómo los distintos componentes interactúan entre sí, asegurando el flujo eficiente de información y la correcta funcionalidad del sistema. En este proyecto, el sistema consta de varios módulos interconectados, que permiten la captura de datos en tiempo real, su procesamiento, visualización y almacenamiento.

El diagrama de arquitectura de alto nivel ilustra la relación entre el dispositivo de captura, el servidor, la interfaz web, y los sensores MEMS integrados en el sistema de monitorización de raquetas. Este enfoque asegura que el sistema esté diseñado para un flujo de información continuo y preciso, con opciones para validación y almacenamiento de datos.

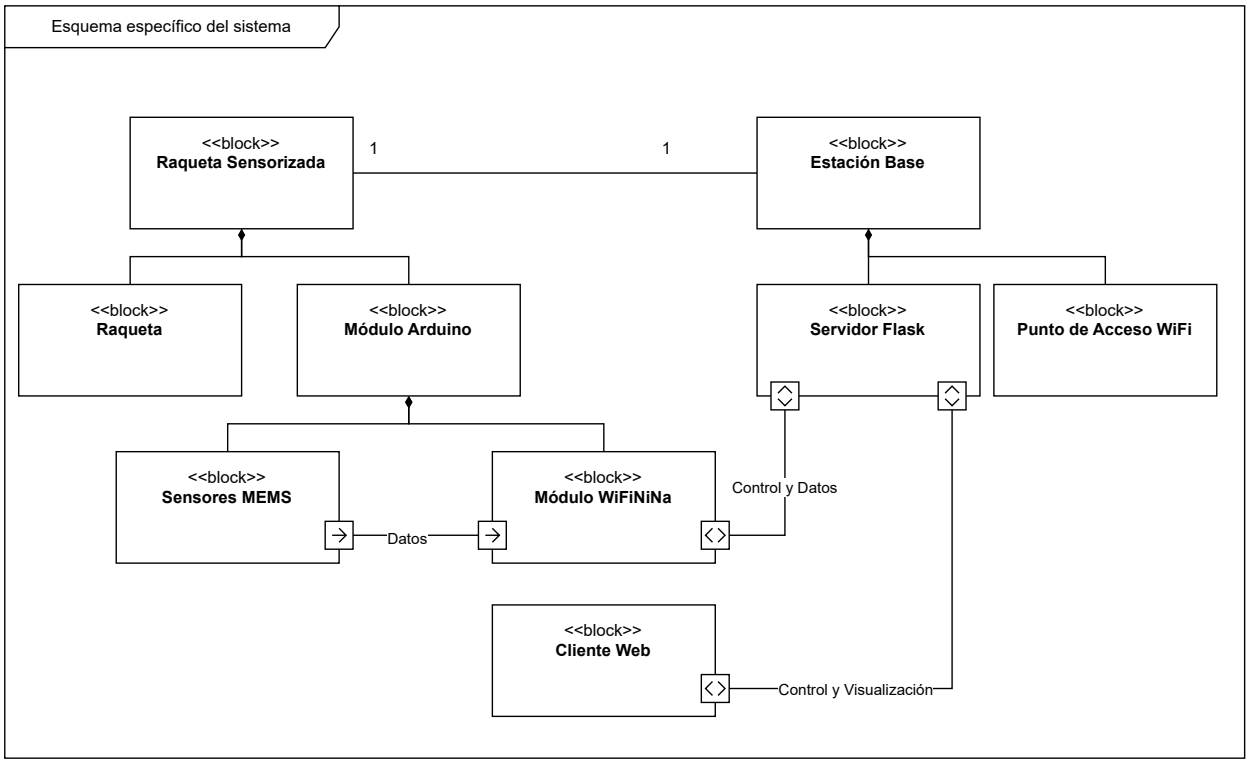

Figura 4.9: Arquitectura específica del sistema

4.4.1. Componentes principales del sistema

El sistema está compuesto por varios componentes interconectados que permiten la captura, procesamiento, visualización y almacenamiento de datos de los sensores MEMS integrados en el Arduino. A continuación, se describen los principales componentes del sistema y su rol en el flujo de información:

- **Arduino Nano IoT 33 con sensores MEMS:**

  El Arduino Nano IoT 33 es el dispositivo de captura principal. Este microcontrolador cuenta con sensores MEMS integrados, específicamente un acelerómetro y un giroscopio, que miden el movimiento de la raqueta en tiempo real. Los datos recogidos por estos sensores se envían al servidor Flask para su procesamiento.

- **Acelerómetro:** Mide la aceleración de la raqueta en los ejes X, Y y Z, proporcionando información sobre la velocidad y la dirección del movimiento.
- **Giroscopio:** Mide la velocidad angular de la raqueta, lo que permite calcular el ángulo de giro y la rotación durante el juego.

- **Servidor Flask:**

  El servidor Flask actúa como el intermediario entre el Arduino y la interfaz web. Este componente es esencial para recibir los datos desde el dispositivo de captura, procesarlos y almacenarlos según sea necesario. Además, el servidor gestiona las solicitudes del usuario enviadas desde el panel web y controla el flujo de comandos hacia el Arduino, como iniciar y detener la captura de datos.

  - **Procesamiento de datos:** El servidor recibe los datos en tiempo real y los prepara para su visualización en la interfaz web.
  - **API para interacciones:** El servidor proporciona una serie de rutas para la comunicación con el panel web, incluyendo el inicio de captura, pruebas controladas, y verificación de los datos.

- **Panel web interactivo:**

  El panel web es la interfaz de usuario que permite interactuar con el sistema. Los usuarios pueden iniciar la captura de datos, realizar pruebas controladas de los sensores, y verificar los datos capturados antes de almacenarlos.

  - **Inicio y parada de captura:** Los usuarios pueden iniciar la captura de datos en tiempo real o de manera temporizada, controlando el tiempo de muestreo directamente desde la interfaz.
  - **Visualización de datos en el momento:** El panel presenta los datos en gráficos interactivos que muestran las mediciones del acelerómetro y el giroscopio en los tres ejes.
  - **Almacenamiento de datos:** Tras la verificación de los datos, el usuario puede optar por guardarlos en un archivo CSV para un análisis posterior.

4.4.2. Comunicación

La comunicación entre los diferentes componentes del sistema es fundamental para asegurar el flujo de datos en tiempo real y permitir que el usuario interactúe eficientemente con el sistema. Este apartado describe cómo se realiza la transmisión de datos desde la raqueta hasta el servidor, y cómo estos datos se presentan en la interfaz web.

- **Transmisión de datos desde la raqueta (Arduino):** La raqueta equipada con el Arduino Nano IoT 33, que cuenta con sensores MEMS (acelerómetro y giroscopio), transmite los datos al servidor Flask a través de una conexión Wi-Fi. Para ello, se utilizan las clases WiFiUDP del chip WiFiNiNa, que permiten una comunicación eficiente. Antes de la transmisión, los datos recogidos por los sensores se procesan en el Arduino, transformándolos de formato float a bytes, optimizando así la velocidad y fiabilidad de la transmisión.

- **Interacción entre el servidor y el panel web:** El servidor Flask actúa como intermediario, recibiendo los datos de la raqueta, y luego transformándolos nuevamente de bytes a float. Además de recibir los datos, Flask gestiona las peticiones enviadas desde el panel web. El panel web se comunica con el servidor Flask mediante peticiones HTTP, enviando comandos como iniciar o detener la captura de datos y recibiendo la información procesada para su visualización.
- **Flujo de datos:** Los datos fluyen desde los sensores MEMS del Arduino hacia el servidor Flask, y desde este último hacia la interfaz web para su visualización. Todo el flujo de información está diseñado para garantizar una comunicación fluida y eficiente en tiempo real, permitiendo al usuario interactuar con los datos a medida que se capturan.

4.4.3. Flujo de datos y procesos

- **Captura de datos:** Los sensores MEMS integrados en el Arduino capturan datos de aceleración y velocidad angular, que son procesados en el dispositivo transformando los valores de float a bytes antes de su transmisión al servidor Flask a través de Wi-Fi.
- **Procesamiento de datos:** El servidor Flask recibe los datos en formato bytes y los convierte nuevamente a float. Una vez recibidos, los datos pasan por un proceso de validación, que incluye verificar si corresponden a las mediciones esperadas. Si los datos son válidos, pueden ser visualizados en el momento y, si el usuario lo decide, almacenados en un archivo CSV para análisis posterior.
- **Visualización en el momento:** La interfaz web está diseñada para presentar los datos a medida que se reciben desde el servidor. Los valores del acelerómetro y giroscopio se muestran en gráficos interactivos que permiten al usuario ver las tendencias de movimiento y rotación de la raqueta en tiempo real.
- **Almacenamiento de datos:** Tras la validación de los datos en el servidor Flask, el usuario tiene la opción de guardarlos en un archivo CSV para su análisis posterior. Esto solo ocurre después de que el sistema haya verificado que los datos son correctos y coinciden con las expectativas.

4.5. Diseño detallado

4.5.1. Diagrama de clases

En el sistema, los componentes principales interactúan para capturar, procesar, transmitir y visualizar los datos de los sensores MEMS. A continuación, se describen las clases y métodos clave del sistema.

1. **Clase Arduino (BinarySendUDP.ino):**
    - **Responsabilidad:** Esta clase se encarga de la captura de datos de los sensores MEMS (acelerómetro y giroscopio), la conversión de estos valores de float a bytes y su posterior transmisión al servidor Flask utilizando UDP.

- **Métodos:**
  - *setup():* Inicializa el Arduino, los sensores y la conexión WiFi utilizando el chip WiFiNiNa.
  - *loop():* El ciclo principal que continuamente lee los datos de los sensores y envía los datos al servidor.
  - *readSensors():* Captura las lecturas de los sensores MEMS (acelerómetro y giroscopio).
  - *sendUDP():* Envía los datos leídos y empaquetados en bytes al servidor Flask a través del protocolo UDP.
  - *packData():* Transforma los datos de float a bytes para optimizar su transmisión por la red.

2. **Clase Servidor UDP (pythonReceiver.py):**
   - **Responsabilidad:** El servidor UDP recibe los datos enviados por el Arduino, los desempaqueta transformando los bytes a float, valida los datos y los almacena o visualiza.
   - **Métodos:**
     - *receiveData():* Recibe los paquetes UDP enviados por el Arduino.
     - *unpackData():* Convierte los datos de bytes a float utilizando *struct.unpack()*.
     - *saveToCsv():* Almacena los datos procesados en un archivo CSV para análisis posterior.
     - *processData():* Procesa los datos recibidos para su visualización en tiempo real o almacenamiento.
3. **Clase Servidor Flask:**
   - **Responsabilidad:** Esta clase gestiona las solicitudes del panel web, permite el inicio o finalización de la captura de datos, y envía los datos procesados al panel web para su visualización o almacenamiento.
   - **Métodos:**
     - *command():* Recibe comandos del panel web para iniciar o detener la captura de datos y transmite estos comandos al Arduino.
     - *testSensors():* Realiza una prueba de sensores por un periodo corto de 10 segundos.
     - *checkData():* Comprueba los datos capturados y los envía al panel web para su visualización.
     - *downloadCsv():* Permite al usuario descargar los datos en formato CSV.
4. **Clase Panel Web (HTML/JavaScript):**
   - **Responsabilidad:** Esta clase proporciona la interfaz gráfica para que el usuario controle la captura de datos y vea los resultados de manera interactiva. El usuario puede iniciar la captura, realizar pruebas y guardar los datos capturados.

- **Métodos:**
  - *sendCommand():* Envia comandos al servidor Flask para iniciar, detener o temporizar la captura de datos.
  - *preScan():* Realiza una prueba de los sensores enviando una solicitud al servidor Flask para una captura de 10 segundos.
  - *postScan():* Comprueba los datos capturados y actualiza los gráficos.
  - *saveFile():* Permite al usuario guardar los datos en un archivo CSV.

4.5.2. Diagrama de secuencia

El flujo de interacción entre los componentes del sistema para la captura y visualización de datos es el siguiente:

1. **Inicio de captura de datos:**
   - **Usuario:** El proceso comienza cuando el usuario accede al panel web y solicita iniciar la captura de datos mediante el botón ÇOMENZAR". Este paso inicia la secuencia de captura y procesamiento de datos.
   - **Panel web (sendCommand):** El panel web envía una solicitud HTTP al servidor Flask para iniciar la captura de datos. Esta solicitud incluye el comando necesario para activar la captura en el sistema.
2. **Detención de la captura de datos:**
   - **Usuario:** El usuario tiene la opción de detener la captura de datos en cualquier momento. Al hacerlo, puede descargar los datos capturados en formato CSV si así lo desea. La acción de detener la captura se realiza a través del panel web, que envía una solicitud al servidor Flask.
3. **Visualización de datos en el panel web:**
   - **Panel web (postScan):** Una vez procesados, el servidor Flask envía los datos al panel web. El panel web actualiza los gráficos en tiempo real con la nueva información, proporcionando al usuario una visualización actualizada de los datos capturados.

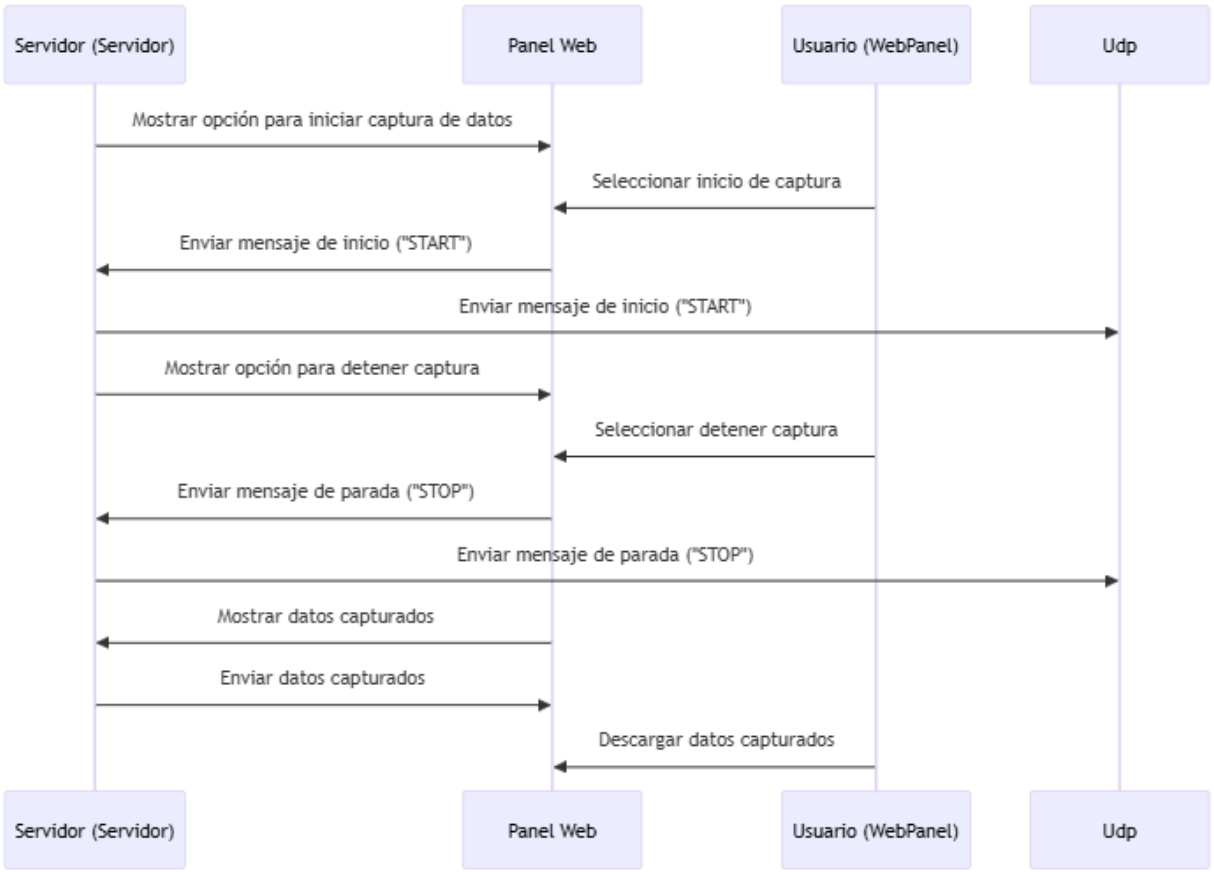

Figura 4.10: Diagrama de interacción panel web - Servidor Flask

4. **Comando al Arduino:**

   - **Servidor Flask (command):** Al recibir la solicitud del panel web, el servidor Flask envía una señal al Arduino para comenzar la captura de datos. Esta señal es retransmitida a través de UDP para que el Arduino inicie el proceso de captura.

5. **Captura y envío de datos por el Arduino:**

   - **Arduino (readSensors):** Una vez que el Arduino recibe el comando de inicio, comienza a capturar datos de los sensores (acelerómetro y giroscopio) mediante la función readSensors(). Los datos obtenidos se empaquetan en formato bytes utilizando la función packData(). Luego, el Arduino envía estos datos al servidor Flask a través de UDP utilizando sendUDP().

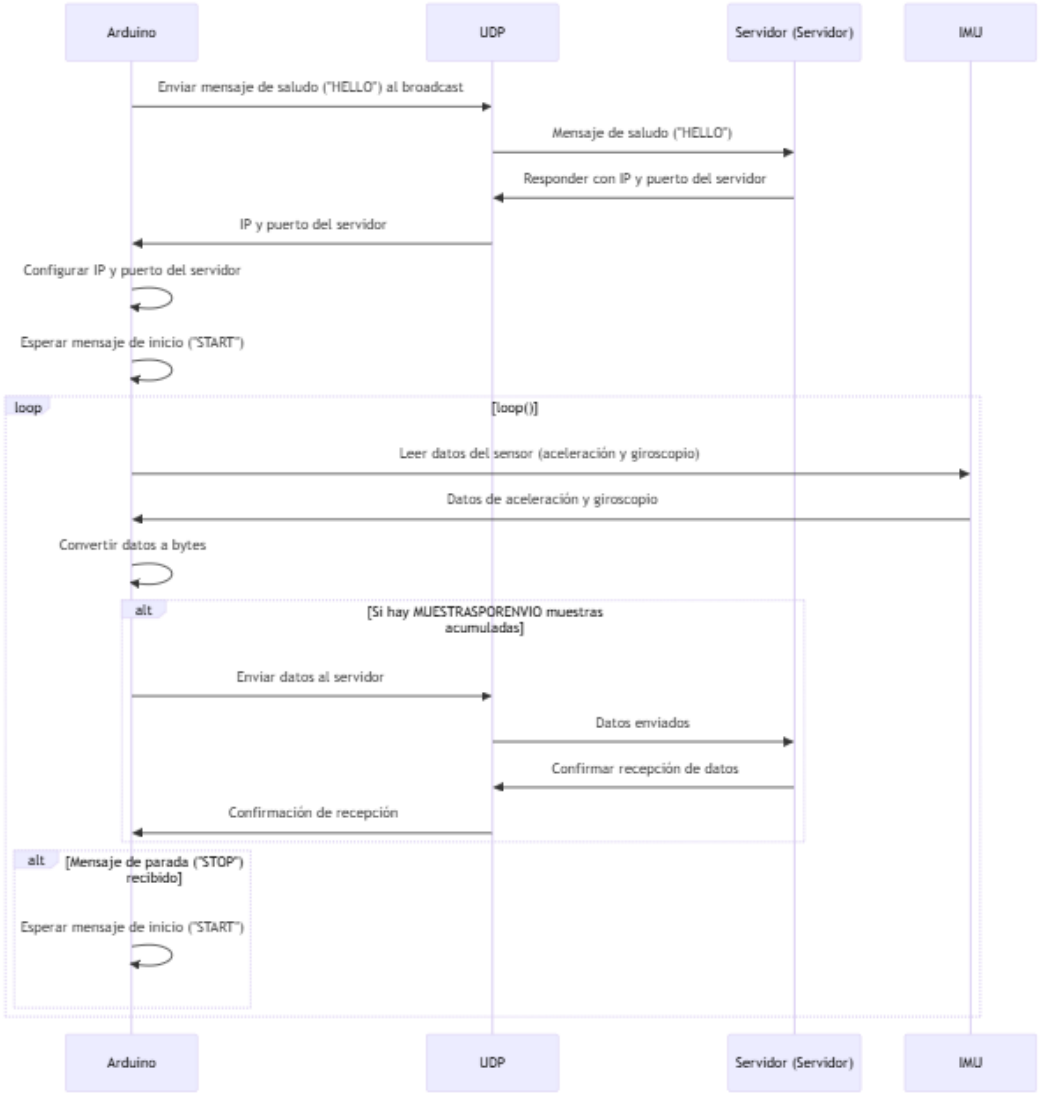

Figura 4.11: Diagrama de interacción servidor Flask - Arduino

6. **Recepción y procesamiento de datos por el servidor Flask:**

   - **Servidor Flask (receiveData)** El servidor Flask recibe los datos en formato bytes enviados por el Arduino. Utiliza la función unpackData() para convertir estos datos de bytes a formato float. Posteriormente, el servidor procesa la información para su visualización o almacenamiento, preparándola para ser enviada al panel web.

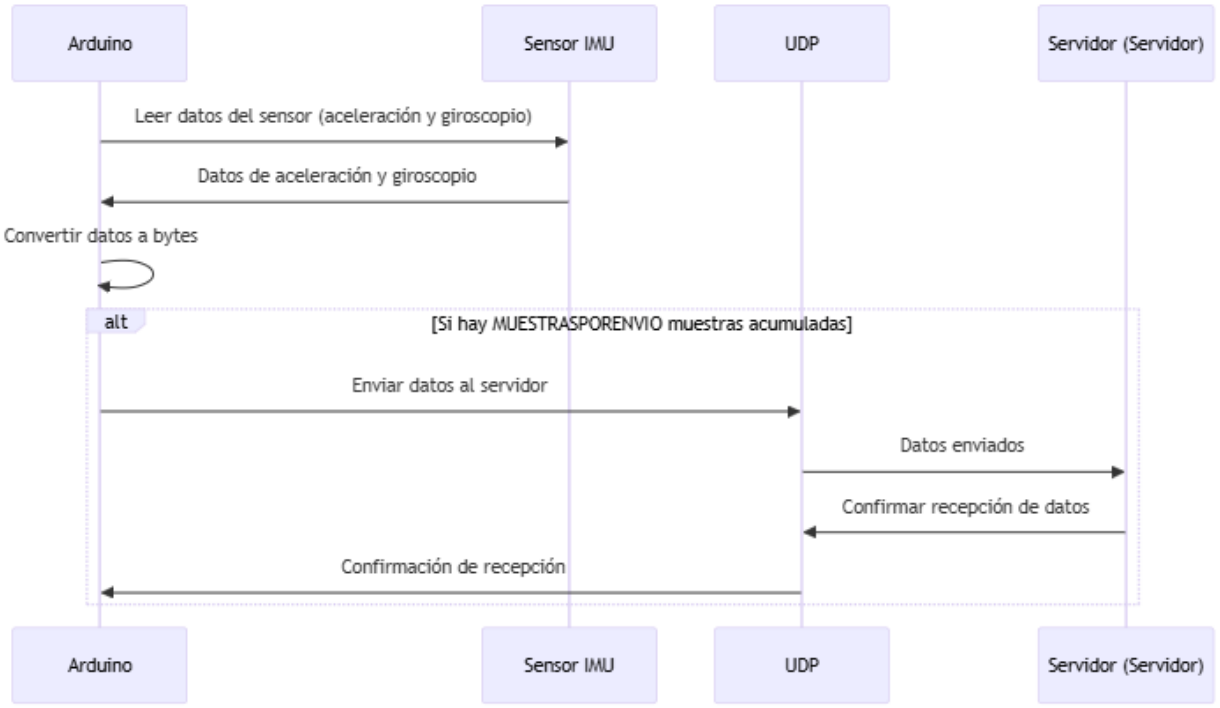

Figura 4.12: Diagrama de procesamiento de datos por el servidor Flask

4.5.3. Interacción entre componentes

El sistema está diseñado para que los componentes trabajen de forma sincronizada, permitiendo la captura, procesamiento y visualización de los datos de los sensores MEMS.

1. **Arduino (BinarySendUDP.ino):**
   - **Captura de Datos (*readSensors*):** Los sensores MEMS capturan datos de aceleración y giroscopio. Estos valores son transformados de float a bytes mediante packData() para optimizar su transmisión.
   - **Transmisión UDP (*sendUDP*):** El Arduino utiliza el protocolo UDP para enviar los datos empaquetados al servidor Flask a través de WiFi.
2. **Servidor Flask (pythonReceiver.py):**

- **Recepión UDP (*receiveData*):** El servidor Flask escucha los paquetes UDP enviados por el Arduino.
- **Procesamiento (*unpackData*):** Utiliza *struct.unpack()* para desempaquetar los datos de bytes a float y proceder a su validación.
- **Almacenamiento o visualización (*saveToCsv / processData*):** Los datos se almacenan en un archivo CSV o se envían al panel web para su visualización en gráficos.

3. **Panel web (HTML/JavaScript):**
   - **Control de captura:** El panel web permite al usuario iniciar y detener la captura de datos, así como realizar pruebas de sensores mediante los botones de control.
   - **Visualización Gráfica (*Chart.js*):**Los datos recibidos del servidor Flask se muestran en gráficos interactivos actualizados en tiempo real mediante la librería *Chart.js*.

4.5.4. Diseño de almacenamiento de datos

Los datos capturados y procesados se almacenan en archivos CSV para su posterior análisis. El flujo es el siguiente:

1. **Captura de datos (*readSensors*):** Los sensores del Arduino capturan datos de aceleración y giroscopio y los envían al servidor Flask.
2. **Procesamiento (*unpackData*):** El servidor transforma los datos de bytes a float, los valida y prepara para el almacenamiento.
3. **Estructura del archivo CSV:** Los datos se guardan en un archivo CSV, donde cada fila contiene:
   - *timestamp:* Marca temporal de la captura.
   - *accelerometerX, accelerometerY, accelerometerZ:* Valores de aceleración.
   - *gyroX, gyroY, gyroZ:* Valores de velocidad angular.
4. **Descarga del usuario (*saveFile*):** El usuario puede descargar el archivo CSV desde el panel web.

4.5.5. Algoritmos y procesamiento de datos

El sistema emplea diversos algoritmos para procesar y validar los datos antes de su almacenamiento o visualización:

1. **Conversión de float a bytes en Arduino (*packData*):** Los datos de los sensores son transformados de float a bytes para reducir el tamaño de los paquetes enviados.
2. **Recepción y conversión en el servidor (*unpackData*):** El servidor Flask convierte los bytes de vuelta a float utilizando *struct.unpack()*, lo que permite procesar los datos para visualización o almacenamiento.

3. **Validación de datos:** Los datos son validados en el servidor antes de ser visualizados o almacenados. Se descartan los valores fuera de rango.

4.5.6. Interfaz gráfica(Panel web)

El panel web ofrece una interfaz intuitiva que permite al usuario controlar la captura y visualizar los datos en el momento:

1. **Botones de control:**
   - **COMENZAR:** Envía el comando para iniciar la captura.
   - **COMENZAR:** Realiza una captura durante un tiempo especificado.
   - **COMENZAR:** Detiene la captura en curso.
2. **Gráficos interactivos:** Los datos capturados se visualizan en gráficos interactivos generados por *Chart.js*. Estos gráficos permiten alternar entre los ejes X, Y y Z del acelerómetro y giroscopio.
3. **Descarga de datos (*saveFile*):** El usuario puede guardar los datos capturados en un archivo CSV

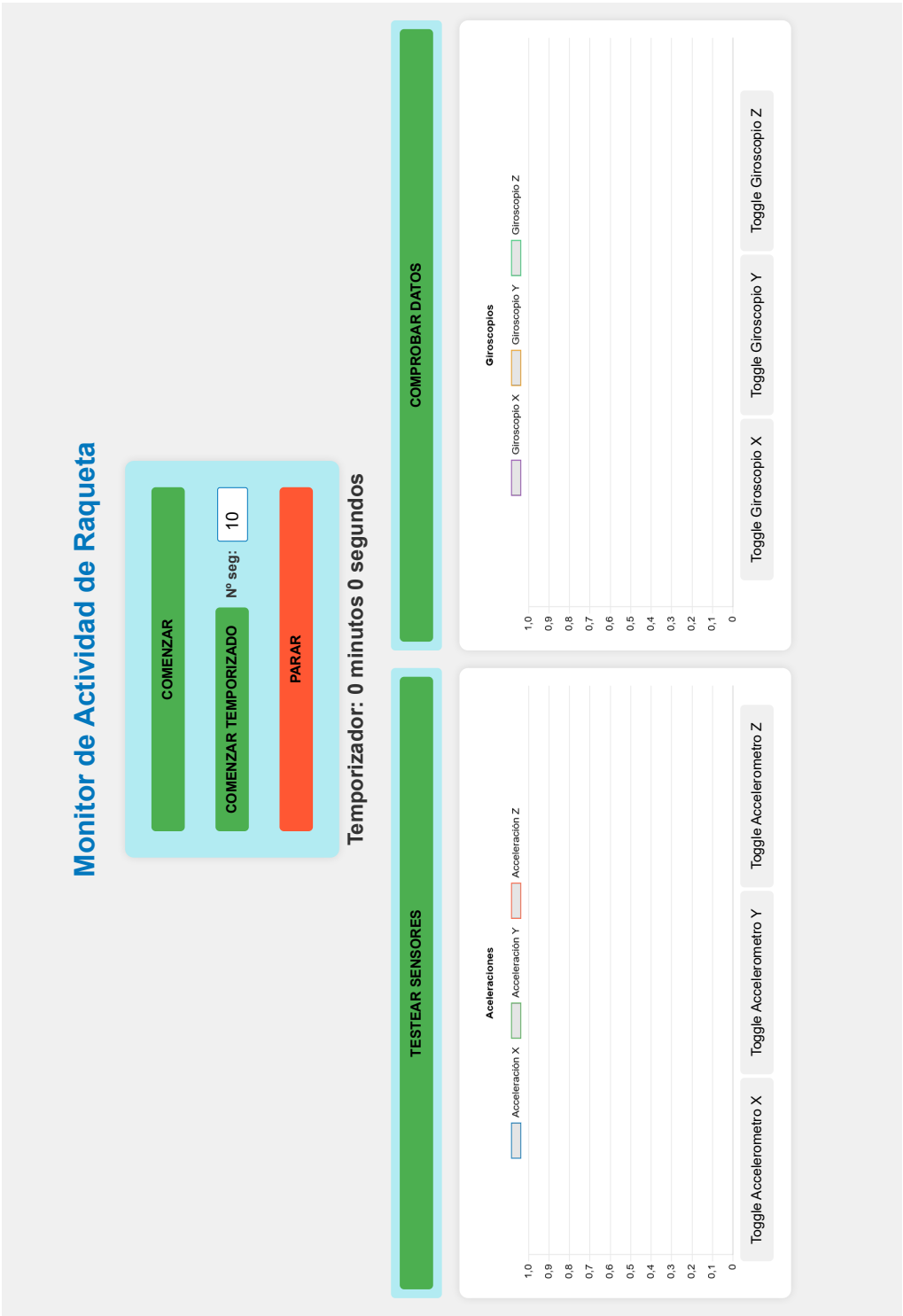

Figura 4.13: Interfaz de usuario

Capítulo 5

Implementación

5.1. Entorno de desarrollo

El entorno de desarrollo para este proyecto comprende tanto los componentes de hardware como las herramientas de software necesarias para la implementación del sistema de adquisición de datos en tiempo real. El objetivo principal fue integrar de manera eficiente los sensores MEMS en una raqueta de bádminton, utilizar una placa Arduino Nano IoT 33 para la captura de datos y establecer un sistema de comunicación entre esta y un servidor Flask. Todo ello permitió la transmisión, procesamiento y visualización de datos en tiempo real. A continuación, se describen los componentes de hardware utilizados, el software empleado y la estructura general del sistema.

5.1.1. Hardware utilizado

En este proyecto, se utilizaron los siguientes componentes de hardware:

- **Arduino Nano IoT 33:** Es la placa principal utilizada para la adquisición de datos. Incorpora sensores MEMS (Microelectromechanical Systems), como un acelerómetro y un giroscopio, que permiten medir la aceleración y la velocidad angular de la raqueta de bádminton. Esta placa fue seleccionada por su capacidad de conectarse a redes Wi-Fi, lo cual es esencial para la transmisión de datos en tiempo real.

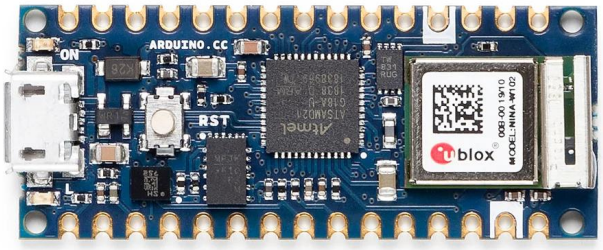

Figura 5.1: Placa Arduino Nano 33 Iot

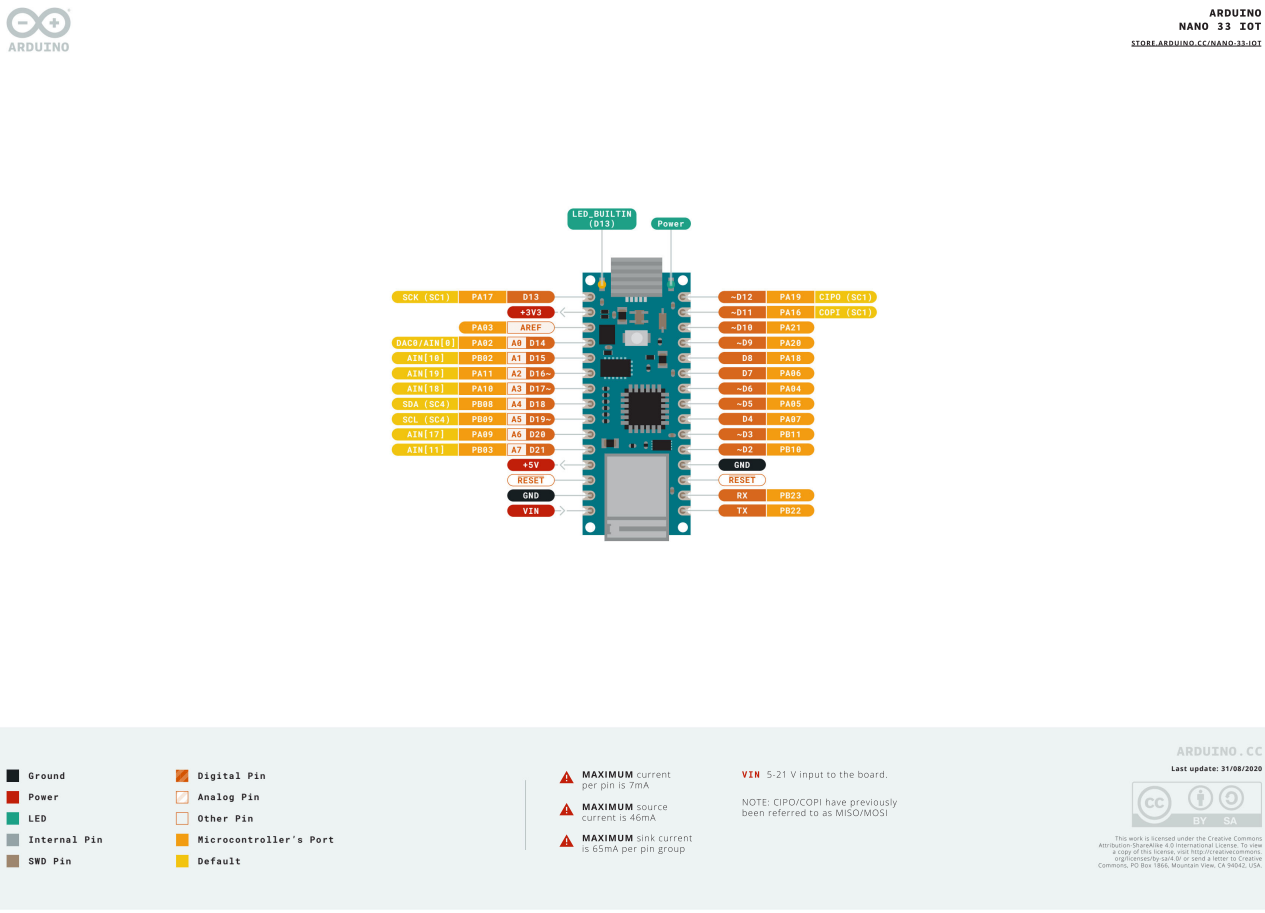

Figura 5.2: Pinout de la placa Arduino Nano 33 Iot

- **Sensores MEMS (Acelerómetro y Giroscopio):** Los sensores MEMS integrados en la placa Arduino Nano IoT 33 proporcionan mediciones en tiempo real de la aceleración y la velocidad angular de la raqueta durante el juego.

- **Raqueta de bádminton:** La raqueta fue modificada para alojar la placa Arduino y los sensores MEMS sin afectar su funcionalidad durante el juego. Se tuvieron en cuenta consideraciones como la ubicación de los sensores para obtener mediciones precisas y mínimamente intrusivas.

5.1.2. Software utilizado

El software juega un papel fundamental en la captura, procesamiento y visualización de los datos. El entorno de desarrollo está compuesto por las siguientes herramientas y tecnologías:

- **Arduino IDE:** Utilizado para programar la placa Arduino Nano IoT 33. Este entorno permite escribir, compilar y cargar el código en la placa. El código fue escrito en C++, utilizando las bibliotecas necesarias para gestionar los sensores MEMS y la conectividad Wi-Fi.
- **bibliotecas de Arduino:**
  - **WiFiNiNa:** Utilizada para habilitar la conectividad Wi-Fi en la placa Arduino Nano IoT 33. Esta biblioteca permite que el Arduino se conecte a una red Wi-Fi local para transmitir datos al servidor Flask.
  - **Arduino_LSM6DS3** Biblioteca que permite la interacción con los sensores MEMS (acelerómetro y giroscopio) integrados en la placa. Esta biblioteca facilita la lectura de datos de los sensores y su procesamiento posterior.

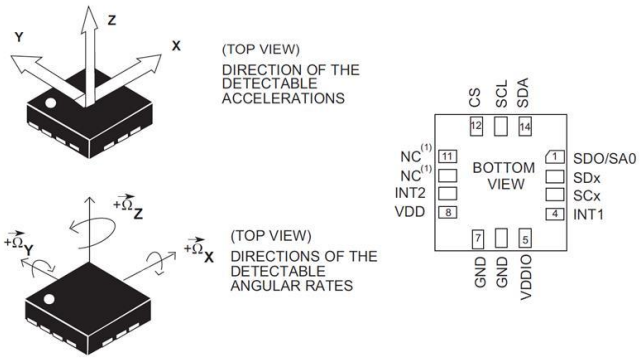

Figura 5.3: Esquema del sensor LSM6DS3

- **Python:** Utilizado para programar el servidor Flask, encargado de recibir, procesar y visualizar los datos enviados por el Arduino. La elección de Python se debe a su simplicidad y la gran cantidad de bibliotecas disponibles para manejar la recepción de datos y la creación de aplicaciones web.
- **Flask:** Un microframework de Python utilizado para implementar el servidor que recibe los datos enviados desde el Arduino. Flask proporciona una interfaz para iniciar y detener la captura de datos, así como para visualizar los resultados en tiempo real.
- **Chart.js:** Una biblioteca de JavaScript utilizada para la visualización de datos en gráficos interactivos dentro del panel web. Los datos capturados por los sensores se presentan en gráficos dinámicos que muestran la aceleración y la velocidad angular de la raqueta en tiempo real.

5.1.3. Estructura del sistema

El sistema está diseñado en tres niveles:

- **Dispositivo de captura:** El Arduino Nano IoT 33 con los sensores MEMS, que toma las mediciones de la raqueta en tiempo real y transmite los datos al servidor.
- **Servidor Flask:** Responsable de la recepción y procesamiento de los datos. Además, gestiona las peticiones del panel web y la visualización de los datos.
- **Interfaz web:** Un panel web accesible para el usuario, que permite interactuar con el sistema, iniciar la captura de datos, y visualizar la información en gráficos en tiempo real.

5.2. Montaje del sistema

En este apartado se detalla el proceso de montaje del sistema, que incluye la configuración del hardware, el ensamblaje de los componentes en la raqueta, y la conexión de los elementos que permiten la captura, procesamiento y visualización de datos en tiempo real.

5.2.1. Preparación del hardware

El sistema utiliza una placa Arduino Nano IoT 33, conectada a un regulador de 5V mediante un cable USB-C. El regulador no solo estabiliza la corriente que recibe la placa, sino que también está conectado a una batería recargable.

- **Circuito de alimentación:**

  La alimentación de la placa se realiza a través de un sistema que incluye un interruptor en la conexión entre el regulador de 5V y el Arduino. Este interruptor permite dos configuraciones:

  - **Carga de la batería:** Cuando el interruptor está en la posición “apagado”, el regulador carga la batería.
  - **Alimentación del Arduino:** Cuando el interruptor está en la posición “encendido”, el circuito se cierra y la batería alimenta directamente al Arduino.

Para realizar esta configuración, se han tenido que soldar los circuitos que conectan el regulador, la batería y la placa Arduino, asegurando la correcta integración de los tres componentes y un flujo de energía estable.

Este diseño permite que el sistema funcione de manera autónoma, sin depender de una conexión constante a una fuente de energía externa, y facilita la carga de la batería cuando el sistema no está en uso.

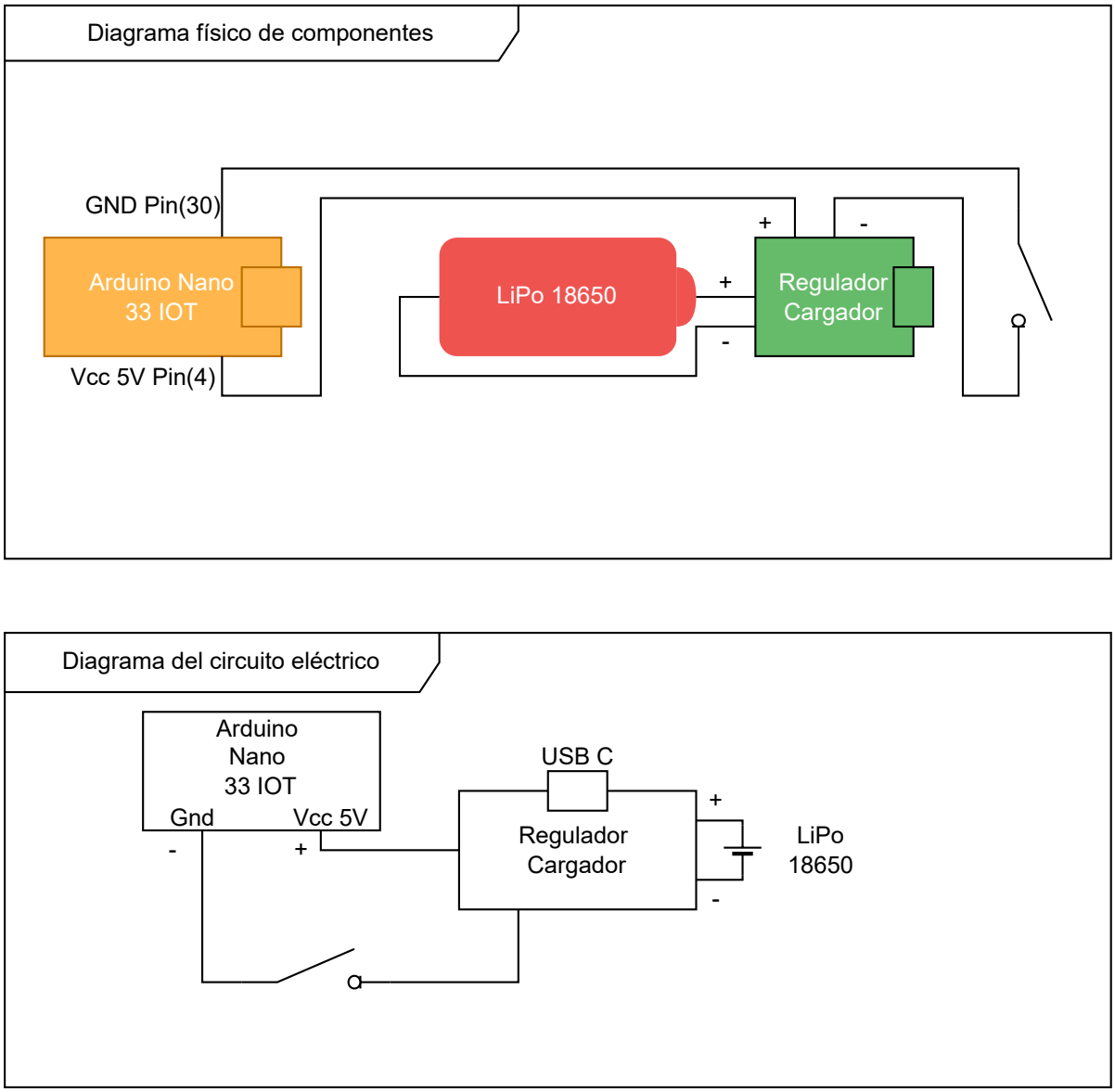

Figura 5.4: Diseño del circuito electrónico

5.2.2. Ensamblaje de la raqueta

El ensamblaje de la raqueta fue un paso crucial en la implementación del sistema, ya que implicaba integrar componentes electrónicos en un dispositivo diseñado para un uso completamente distinto, sin comprometer la funcionalidad ni la integridad estructural de la raqueta. A continuación, se detallan los pasos seguidos y las consideraciones tomadas en cuenta para realizar el ensamblaje.

Selección de la raqueta y creación del mango La elección de la raqueta de bádminton fue un factor importante, ya que la incorporación de los componentes electrónicos no debía alterar su peso o balance significativamente. Para optimizar la integración, se optó por diseñar y fabricar el mango de la raqueta mediante impresión 3D. El material utilizado fue translúcido para interferir lo mínimo posible con la transmisión de datos de los sensores MEMS al servidor Flask a través de Wi-Fi. El uso de la impresión 3D permitió personalizar el diseño del mango para que pudiera alojar adecuadamente los componentes electrónicos, asegurando a la vez la ergonomía del dispositivo para que el jugador pudiera utilizarlo sin notar diferencia en el manejo de la raqueta.

Preparación de los componentes

Antes del ensamblaje, se prepararon los siguientes componentes:

- **Arduino Nano IoT 33:** Esta placa fue seleccionada por su tamaño compacto y su capacidad para gestionar la conexión Wi-Fi, esencial para la transmisión de datos en tiempo real.
- **Sensores MEMS:** Integrados en la placa Arduino, estos sensores miden la aceleración y la velocidad angular de la raqueta durante el juego.
- **Regulador de 5V y batería:** Se utilizó un regulador de 5V conectado a una batería recargable para asegurar una alimentación estable y garantizar que el sistema funcionara de manera autónoma.

Diseño del circuito de alimentación

El sistema de alimentación se diseñó para permitir un uso continuo, asegurando que el Arduino esté encendido únicamente cuando se requiere la captura de datos. Se incorporó un interruptor entre el regulador de 5V y la batería para permitir dos configuraciones:

- **Modo de carga:** Con el interruptor en .ªpagado", la batería se carga a través del regulador de 5V.
- **Modo operativo:** En .ençendido", el circuito se cierra y la batería alimenta al Arduino.

Las conexiones entre el regulador, la batería y la placa Arduino se soldaron para asegurar la durabilidad y fiabilidad del sistema durante los movimientos intensos del juego de bádminton.

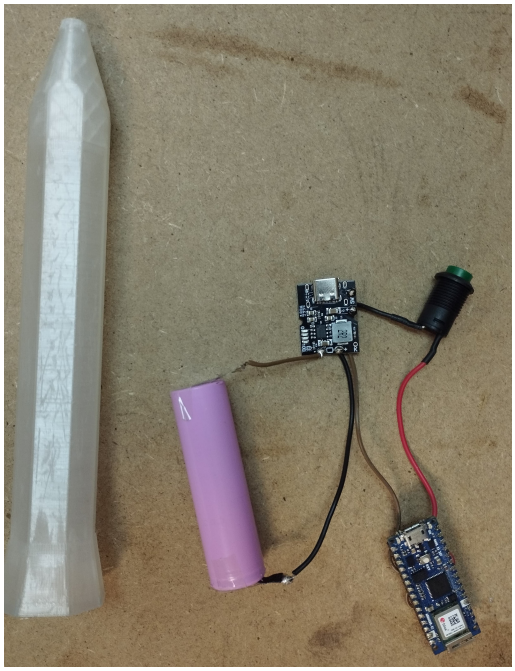

Figura 5.5: Sistema ensamblado

Integración de los componentes en el mango

Una vez preparados los componentes electrónicos, se procedió a su integración en el mango impreso en 3D. Este mango fue diseñado específicamente para acomodar de manera segura la placa Arduino, los sensores MEMS y la batería, manteniendo un diseño ergonómico y ligero. El material translúcido utilizado en la impresión 3D permitía minimizar la interferencia con las señales Wi-Fi, lo que garantizaba una transmisión de datos eficiente y sin obstáculos.

Figura 5.6: Sistema empotrado en el mango

5.3. Captura y procesamiento de datos

5.3.1. Código de configuración y variables iniciales

El código implementado en el Arduino se encarga de capturar datos de un sensor IMU y transmitirlos a un servidor a través de una red WiFi. Utiliza las bibliotecas `WiFiNINA`, `Arduino_LSM6DS3`, y `WiFiUdp` para gestionar la conexión WiFi, la lectura de datos del sensor IMU, y la comunicación UDP, respectivamente. A continuación, se detalla la estructura y funcionamiento del código:

```
#include <WiFiNINA.h>
#include <Arduino_LSM6DS3.h>
#include <WiFiUdp.h>

// Credenciales de Red
const char* ssid = "LAPTOP-NICO";
const char* password = "84|4Z98a";

// Mensajes de Protocolo de Comunicación
const char* startMessage = "START";
const char* stopMessage = "STOP";
const char* handshakeMessage = "HELLO";
```

Código Fuente 5.1: Arduino bibliotecas y definición de constantes

Desglose de la implementación

1. **Inclusión de bibliotecas:**
   - `#include <WiFiNINA.h>`: Incluye la biblioteca para gestionar la conexión Wi-Fi y las operaciones relacionadas con la red en la placa Arduino. Es compatible con la placa Arduino Nano 33 IoT y permite al dispositivo conectarse a redes inalámbricas y enviar o recibir datos.
   - `#include <Arduino_LSM6DS3.h>`: Esta biblioteca gestiona la interacción con el sensor LSM6DS3, que incluye un acelerómetro y un giroscopio integrados en la placa. Permite capturar datos de movimiento en tres dimensiones.
   - `#include <WiFiUdp.h>`: Proporciona las funciones necesarias para enviar y recibir paquetes de datos a través del protocolo UDP (User Datagram Protocol). UDP es ideal para la transmisión rápida de datos en tiempo real.
2. **Credenciales de red:**
   - `const char* ssid = "LAPTOP-NICO";`: Define el nombre de la red Wi-Fi (SSID) a la que el dispositivo se conectará. En este caso, la red se llama `"LAPTOP-NICO"`.

- `const char* password = "84|4Z98a";`: Define la contraseña de la red Wi-Fi. Es necesaria para autenticar y conectar el dispositivo a la red inalámbrica.

  Estas credenciales son fundamentales para establecer la conexión inicial a la red. Sin ellas, el dispositivo no puede conectarse y, por tanto, no puede enviar ni recibir datos.

3. **Mensajes de protocolo de comunicación:**

   - `const char* startMessage = "START";`: Define el mensaje que se utiliza para iniciar la captura de datos. Cuando el dispositivo recibe este mensaje del servidor, comienza a capturar y enviar datos de los sensores.
   - `const char* stopMessage = "STOP";`: Define el mensaje que se utiliza para detener la captura de datos. Al recibir este mensaje, el dispositivo deja de capturar y enviar datos.
   - `const char* handshakeMessage = "HELLO";`: Define un mensaje de saludo o confirmación que se puede utilizar para verificar la conexión entre el dispositivo y el servidor. Es útil para realizar un "handshake" inicial que confirme que ambos están listos para comunicarse.

Propósito de la configuración:

1. **Establecimiento de la conexión Wi-Fi:**

   - La inclusión de las credenciales de red permite al dispositivo conectarse a la red Wi-Fi específica. Esto es un paso esencial para habilitar la comunicación inalámbrica y la transmisión de datos.

2. **Protocolo de control de captura de datos:**

   - Los mensajes `startMessage` y `stopMessage` permiten el control remoto de la captura de datos del dispositivo. Estos comandos se envían desde el servidor y el dispositivo responde iniciando o deteniendo la captura de datos según corresponda.
   - Este tipo de control es crucial en aplicaciones de monitoreo en tiempo real, donde se necesita iniciar o detener la captura de datos basándose en eventos específicos o condiciones predefinidas.

3. **Mecanismo de verificación de conexión:**

   - El mensaje `handshakeMessage ("HELLO")` puede utilizarse para verificar que la conexión entre el dispositivo y el servidor está activa antes de iniciar la transmisión de datos. Este mecanismo de verificación es útil para asegurarse de que ambas partes están listas para intercambiar datos.

```
// La dirección IP y el puerto del servidor Python
IPAddress serverIP;
unsigned int serverPort;

// Inicializar la instancia de la biblioteca UDP
WiFiUDP Udp;

// Variables Globales Estáticas
#define DATOSPORMUESTRA 7
#define BYTESPORMUESTRA 28
#define MUESTRASPORENVIO 20
#define PACKETSIZE (MUESTRASPORENVIO * BYTESPORMUESTRA)
#define UDPPORT 12345

// Contadores
int contadorMuestras = 0;
int startNumber = 0;
long tiempoInicioCaptura = 0;
byte* byteArrayEnvio;
```

Código Fuente 5.2: Arduino inicialización de variables

Desglose de la implementación

1. **Configuración de la dirección IP y el puerto del servidor:**
   - `IPAddress serverIP;`: Declara una variable de tipo `IPAddress` para almacenar la dirección IP del servidor Python al que el dispositivo se conectará. La dirección IP se asignará en otra parte del código.
   - `unsigned int serverPort;`: Declara una variable de tipo `unsigned int` para almacenar el puerto del servidor. Este puerto será el punto de acceso a través del cual se enviarán los datos.

     Estas variables son esenciales para dirigir correctamente los paquetes de datos al destino adecuado en la red.
2. **Inicialización de la instancia de la biblioteca UDP:**
   - `WiFiUDP Udp;`: Crea una instancia de la clase `WiFiUDP`, que proporciona las funciones necesarias para enviar y recibir paquetes a través del protocolo UDP. Esta instancia se utiliza para gestionar la comunicación de datos con el servidor.
3. **Definición de variables globales estáticas:**

- `#define DATOSPORMUESTRA 7`: Define el número de valores de sensor que se capturan por cada muestra. En este caso, cada muestra contiene 7 valores de datos diferentes, que pueden incluir lecturas de acelerómetro, giroscopio, etc.
- `#define BYTESPORMUESTRA 28`: Define el tamaño de cada muestra en bytes. Como cada valor de sensor se representa como un `float` de 4 bytes y hay 7 valores en cada muestra, el tamaño total de cada muestra es de 28 bytes (`7 valores * 4 bytes = 28 bytes`).
- `#define MUESTRASPORENVIO 20`: Define el número de muestras que se acumulan antes de enviarlas al servidor. En este caso, se acumulan 20 muestras antes de enviar un paquete.
- `#define PACKETSIZE (MUESTRASPORENVIO * BYTESPORMUESTRA)`: Calcula el tamaño total del paquete de datos que se enviará al servidor. El tamaño del paquete es el producto del número de muestras por envío y el tamaño de cada muestra (560 bytes en total).
- `#define UDPPORT 12345`: Define el puerto UDP local en el que el dispositivo escuchará o enviará paquetes. Este puerto se utiliza para la comunicación entrante y saliente de datos.

4. **Contadores:**

   - `int contadorMuestras = 0;`: Contador que lleva el número de muestras acumuladas antes de enviar un paquete. Se incrementa cada vez que se captura una muestra y se reinicia cuando se envía un paquete.
   - `int startNumber = 0;`: Variable que puede utilizarse para controlar el inicio de la captura de datos. Podría representar un indicador del estado de la captura o del número de inicios de captura.
   - `long tiempoInicioCaptura = 0;`: Variable que almacena el tiempo de inicio de la captura de datos. Se utiliza en conjunto con millis() para medir la duración de la captura.

5. **Puntero de buffer de envío:**

   - `byte* byteArrayEnvio;`: Declara un puntero de tipo `byte` que apunta al array `byteArrayEn` Este puntero se utiliza para gestionar dinámicamente el almacenamiento y envío de los datos acumulados.

Propósito de la configuración:

1. **Preparación para la captura y envío de datos:**

   - Las variables y buffers definidos permiten capturar datos de los sensores, convertirlos en bytes y acumularlos en un buffer de envío. Esto es crucial para enviar datos en paquetes grandes, minimizando el número de transmisiones y aumentando la eficiencia.

2. **Control de la comunicación UDP:**

- La instancia `Udp` y las variables de dirección IP y puerto aseguran que los paquetes se envían al servidor correcto y en el formato adecuado. El uso de un puerto UDP específico facilita la gestión de las conexiones de red y asegura que los datos sean enviados con mayor facilidad y velocidad.

3. **Control de la frecuencia de envío:**

   - Los contadores permiten controlar cuántas muestras se acumulan antes de enviar un paquete. Esto es importante para regular la frecuencia de envío de datos y evitar saturar la red con demasiados paquetes pequeños.

5.3.2. Inicialización del sistema

Durante la fase de inicialización, el Arduino se conecta a una red Wi-Fi y configura los sensores MEMS. El código relevante para esta fase es el siguiente:

```
void setup() {
    Serial.begin(9600);
    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED) {
        delay(100);
        WiFi.begin(ssid, password);
    }

    Udp.begin(UDPPORT);

    if (!IMU.begin()) {
        while (1);
    }

    sendBroadcastMessage();

    waitForStartMessage();
}
```

Código Fuente 5.3: Arduino inicialización del sistema

Desglose de implementación

1. **Inicio de la comunicación serial:**

   - `Serial.begin(9600);`: Inicia la comunicación serie con una velocidad de transmisión de 9600 baudios. Esta comunicación se utiliza para depuración y monitoreo del estado del

dispositivo a través del puerto serie. Es útil para verificar el estado de la conexión y la configuración del sensor.

2. **Conexión a la red WiFi:**

   - `WiFi.begin(ssid, password);`: Inicia el proceso de conexión a la red WiFi utilizando las credenciales (`ssid` y `password`) definidas previamente. Este paso es esencial para establecer la comunicación entre el dispositivo y el servidor a través de la red local.
   - `while (WiFi.status() != WL_CONNECTED) { ... }`: Este bucle `while` se ejecuta hasta que el dispositivo se conecta a la red WiFi. Comprueba continuamente el estado de la conexión (`WL_CONNECTED`) y, si no está conectado, espera 100 milisegundos y reintenta la conexión llamando de nuevo a `WiFi.begin(ssid, password);`.
     - **Propósito de la repetición:** El bucle asegura que el dispositivo no continúe con el resto de la configuración hasta que se haya establecido correctamente la conexión a la red WiFi. Este enfoque es robusto frente a fallos temporales de conexión.

3. **Inicialización de la comunicación UDP:**

   - `Udp.begin(UDPPORT);`: Inicializa el puerto UDP local especificado por UDPPORT (en este caso, 12345). Este puerto se utiliza para enviar y recibir paquetes de datos. La inicialización asegura que el dispositivo está listo para gestionar la comunicación UDP con el servidor.

4. **Inicialización del sensor inercial (IMU):**

   - `if (!IMU.begin()) while (1);` : Intenta inicializar el sensor IMU (Inertial Measurement Unit) utilizando la función `IMU.begin()`. Si la inicialización falla (es decir, `IMU.begin()`) devuelve false), el dispositivo entra en un bucle infinito `while (1)`; que detiene la ejecución del código. Esto asegura que no se proceda con el envío de datos si los sensores no están operativos.
     - **Importancia de la verificación:** Esta verificación es crucial, ya que garantiza que el sistema solo continúe si los sensores están funcionando correctamente. De lo contrario, se evita el envío de datos incorrectos o no válidos.

5. **Envío de mensaje de broadcast:**

   - `sendBroadcastMessage();`: Llama a una función (presumiblemente definida en otra parte del código) para enviar un mensaje de broadcast. El propósito de este mensaje es notificar al servidor o a otros dispositivos en la red que el Arduino está operativo y listo para comenzar la captura de datos.
     - **Funcionalidad del broadcast:** El mensaje de broadcast permite al servidor conocer la presencia del dispositivo en la red sin necesidad de conocer su dirección IP específica. Esto facilita la sincronización inicial entre el servidor y el dispositivo.

6. **Espera de mensaje de inicio:**

   - `waitForStartMessage();`: Llama a una función que espera recibir un mensaje de inicio (`START`) desde el servidor. Esta función probablemente monitorea la llegada de paquetes UDP hasta que recibe el mensaje esperado.

     - **Propósito de la espera activa:** Esta espera garantiza que el dispositivo no comience a capturar y enviar datos antes de recibir la instrucción específica del servidor. Esto permite un control remoto sobre el inicio de la captura de datos, asegurando que el sistema solo actúe bajo la dirección del servidor.

Propósito de la configuración inicial

1. **Preparación para la comunicación:**

   - La configuración inicial de la comunicación serie y la conexión WiFi son fundamentales para establecer el enlace de datos entre el dispositivo y el servidor. Sin esta configuración, el dispositivo no podría enviar ni recibir datos.

2. **Verificación del sensor:**

   - La comprobación de la inicialización del IMU asegura que los sensores están listos para capturar datos precisos. Esto es crucial para la fiabilidad del sistema y la calidad de los datos capturados.

3. **Coordinación con el servidor:**

   - El envío del mensaje de broadcast y la espera del mensaje de inicio permiten una sincronización adecuada entre el dispositivo y el servidor, asegurando que la captura de datos comience en el momento adecuado y bajo control remoto.

5.3.3. Captura de datos

La captura de datos se realiza en el bucle principal `loop()`. A continuación, se muestra el código que maneja esta tarea:

```
void loop() {
    if (startNumber == 0 ||
        startNumber * 1000 > (millis() - tiempoInicioCaptura)) {
        float* sensorData =
            (float*)(malloc(DATOSPORMUESTRA * sizeof(float)));
        byte* byteArray =
            (byte*)(malloc(BYTESPORMUESTRA * sizeof(byte)));
        accelerometerPlusgyroscope(sensorData);
        floatToByte(sensorData, byteArray);
        free(sensorData);

        if (contadorMuestras == 0) {
            byteArrayEnvio = (byte*)(malloc(PACKETSIZE * sizeof(byte)));
            copyArrays(byteArrayEnvio, byteArray, contadorMuestras);
            contadorMuestras ++;
        } else {
            sendSamples(ByteArrayEnvio, byteArray, contadorMuestras);
        }

        free(byteArray);
    }
}
```

Código Fuente 5.4: Arduino captura de datos

Desglose de implementación

1. **Condición de captura de datos:**

```
if (startNumber == 0 ||
startNumber * 1000 > (millis() - tiempoInicioCaptura))
```

Esta condición comprueba si el sistema debe iniciar o continuar la captura de datos. Se asegura de que:

- `startNumber == 0`: Si `startNumber` es `0`, la captura de datos se realiza indefinidamente, sin límite de tiempo.
- `startNumber * 1000 >(millis() - tiempoInicioCaptura)`: Si `startNumber` es mayor que `0`, se compara el tiempo transcurrido `(millis() - tiempoInicioCaptura)` con el tiempo de captura especificado `(startNumber * 1000` en milisegundos). Si el tiempo transcurrido es menor que el especificado, la captura continúa.

2. **Asignación dinámica de memoria para datos del sensor:**

```
float* sensorData =
    (float*)(malloc(DATOSPORMUESTRA * sizeof(float)));
byte* byteArray =
    (byte*)(malloc(BYTESPORMUESTRA * sizeof(byte)));
```

   Se asigna memoria dinámica para almacenar temporalmente:

   - `sensorData`: Array de `float` que contendrá los datos del sensor, con un tamaño basado en el número de datos por muestra (`DATOSPORMUESTRA`).
   - `byteArray`: Array de `byte` que contendrá la representación en bytes de los datos del sensor, con un tamaño basado en el número de bytes por muestra (`BYTESPORMUESTRA`).

3. **Captura de datos de sensores:**

   `accelerometerPlusgyroscope(sensorData);`

   La función `accelerometerPlusgyroscope(sensorData)` obtiene los datos del acelerómetro y giroscopio y los almacena en el array `sensorData`. Esta función no se muestra aquí, pero se encarga de llenar el array con los valores actuales de los sensores.

4. **Conversión de datos a bytes:**

   `floatToByte(sensorData, byteArray);`

   La función `floatToByte(sensorData, byteArray)` convierte el array de datos del sensor (`sensorData`) a un array de bytes (`byteArray`). Esta conversión es necesaria para optimizar la transmisión de datos a través de UDP.

5. **Liberación de memoria para datos del sensor:**

   `free(sensorData);`

   Se libera la memoria dinámica asignada al array `sensorData` después de su uso, para evitar fugas de memoria y optimizar el uso de recursos del sistema.

6. **Gestión del buffer de envío:**

```
if (contadorMuestras == 0) {
    byteArrayEnvio = (byte*)(malloc(PACKETSIZE * sizeof(byte)));
    contadorMuestras ++;
} else {
    sendSamples(byteArrayEnvio, byteArray, contadorMuestras);
}
```

- **Inicialización del buffer:** Si `contadorMuestras` es `0`, indica que no hay muestras acumuladas en el buffer de envío:
  - Se asigna memoria dinámica para `byteArrayEnvio` con un tamaño definido por `PACKETSIZE` (número total de bytes a enviar).
  - Se copia la muestra actual (`byteArray`) al buffer de envío (`byteArrayEnvio`) en la posición correspondiente utilizando `copyArrays(byteArrayEnvio, byteArray, contadorMuestras)`.
- **Envío de datos:** Si `contadorMuestras` es distinto de `0`, significa que ya hay muestras acumuladas en el buffer:
  - Se llama a `sendSamples(byteArrayEnvio, byteArray, contadorMuestras)`, enviando las muestras acumuladas al servidor.
  - Después de enviar, `contadorMuestras` se reinicia y el buffer se libera para la próxima acumulación de datos.

7. **Liberación de memoria para el array de bytes:**

   `free(byteArray);`

   Al finalizar la captura y conversión de la muestra actual, se libera la memoria asignada para `byteArray`, evitando así fugas de memoria.

Propósito de la enumeración

1. **Gestión dinámica de recursos:**
   - La asignación y liberación de memoria dinámica (`malloc` y `free`) para las muestras individuales y el buffer de envío permite un uso eficiente de la memoria del Arduino. Esto es especialmente importante dado el limitado espacio de memoria de estos dispositivos.
2. **Control de captura de datos:**
   - La condición `if (startNumber == 0 || startNumber * 1000 >(millis() - tiempoInicioCaptura))` asegura que los datos solo se capturan mientras la captura está activa, proporcionando control sobre la duración de la misma, ya sea indefinida (`startNumber == 0`) o limitada (`startNumber >0`).
3. **Flexibilidad y adaptabilidad del sistema:**
   - La estructura del bucle permite que el sistema se adapte a diferentes condiciones de captura y transmisión, facilitando la implementación de diferentes políticas de control y sincronización con el servidor.

5.3.4. Funciones auxiliares

Las funciones auxiliares manejan la conversión de datos y el envío al servidor:

- **Conversión float a byte**

```
/**
 * @brief Convierte un array de floats a un array de bytes.
 *
 * @param floatToTransform Array de floats a convertir.
 * @param byteArray Array de bytes que almacenará los datos convertidos.
 */
void floatToByte(float* floatToTransform, byte* byteArray) {
  byte* byteArray1 = (byte*)&floatToTransform[0];
  byte* byteArray2 = (byte*)&floatToTransform[1];
  byte* byteArray3 = (byte*)&floatToTransform[2];
  byte* byteArray4 = (byte*)&floatToTransform[3];
  byte* byteArray5 = (byte*)&floatToTransform[4];
  byte* byteArray6 = (byte*)&floatToTransform[5];
  byte* byteArray7 = (byte*)&floatToTransform[6];
  for (int i = 0; i < 4; i++) {
    byteArray[i] = byteArray1[i];
        byteArray[4 + i] = byteArray2[i];
        byteArray[8 + i] = byteArray3[i];
        byteArray[12 + i] = byteArray4[i];
        byteArray[16 + i] = byteArray5[i];
        byteArray[20 + i] = byteArray6[i];
        byteArray[24 + i] = byteArray7[i];
  }
}
```

Código Fuente 5.5: Arduino conversión de float a byte

El principal objetivo de la función es transformar cada valor flotante en su representación en bytes, para así facilitar el envío de los datos por medio de protocolos que no admiten directamente tipos de datos más complejos como los float. Esta conversión permite optimizar el proceso de transmisión y reducir posibles errores de interpretación al transferir datos entre dispositivos.

Desglose de la implementación

1. **Parámetros de la función:**
   - `floatToTransform`: Es un puntero al array de floats que se desea convertir. Cada valor flotante ocupa 4 bytes en memoria.

- `byteArray`: Es el puntero al array de bytes donde se almacenarán los valores convertidos. Este array deberá tener una longitud de `4 * n`, donde `n` es el número de floats en `floatToTransform`.

2. **Asignación de punteros:**
   - Para cada float en `floatToTransform` (desde `floatToTransform[0]` hasta `floatToTransform[6]`), se crea un puntero (`byteArray1, byteArray2, ..., byteArray7`) que apunta al inicio de su representación en bytes. Esto se logra con el operador de conversión `(byte*)&floatToTransform[i]`, que interpreta la dirección del float como un puntero a bytes.
3. **Copiado de bytes:**
   - El array de bytes de destino (`byteArray`) se llena en bloques de 4 bytes, correspondientes a cada float. Se utiliza un bucle `for` para copiar los 4 bytes de cada float desde su puntero correspondiente (`byteArray1, byteArray2, etc.`) al array `byteArray`.
   - Cada ciclo del bucle `for` transfiere 4 bytes de `byteArrayX[i]` a `byteArray[posición+i` donde posición es un múltiplo de 4.
4. **Distribución en el array de bytes:**
   - El array `byteArray` resultante contendrá la representación en bytes de todos los valores flotantes consecutivamente. Por ejemplo, si `floatToTransform` contiene 7 valores flotantes, `byteArray` almacenará 28 bytes (`7 floats x 4 bytes`).

- **Copia de arrays:**

```
/**
 * @brief Copia los valores de un array de bytes a otro,
 * considerando el número de muestras acumuladas.
 *
 * @param arrayDestino
 * Array de destino donde se copiarán los valores.
 * @param arrayOrigen
 * Array de origen desde donde se copiarán los valores.
 * @param numeroMuestrasAcumuladas
 * Número de muestras ya acumuladas en array1.
 */
void copyArrays(byte* arrayDestino, byte* arrayOrigen,
int numeroMuestrasAcumuladas) {
  for (int i = 0; i < BYTESPORMUESTRA; i++) {
    arrayDestino [i + BYTESPORMUESTRA * numeroMuestrasAcumuladas] =
arrayOrigen [i];
  }
}
```

Código Fuente 5.6: Arduino copia de arrays

La función tiene como objetivo transferir un conjunto de datos de una muestra desde `arrayOrigen` a una posición específica dentro de `arrayDestino`. La posición dentro del array de destino se calcula en función del número de muestras ya acumuladas, asegurando que los datos de cada muestra se coloquen de manera contigua en el array de destino.

Desglose de la implementación

1. **Parámetros de la función:**
   - `arrayDestino`: Es el array de bytes donde se almacenarán las muestras copiadas. Este array debe tener suficiente espacio para contener todas las muestras acumuladas.
   - `arrayOrigen`: Es el array de bytes que contiene los datos de la muestra actual que se desea copiar. Normalmente, este array contiene una única muestra de datos en cada llamada a la función.
   - `numeroMuestrasAcumuladas`: Es el número de muestras que ya han sido copiadas en arrayDestino. Se utiliza para calcular la posición de inicio donde se copiará la nueva muestra en el array de destino.
2. **Bucle for:**
   - La estructura del bucle `for` recorre cada elemento del `arrayOrigen`, que contiene los datos de la muestra a copiar.

- El índice `i` se utiliza para acceder a cada byte de `arrayOrigen`, y el valor `i + BYTESPORMUESTRA * numeroMuestrasAcumuladas` determina la posición exacta en `arrayDestino` donde se almacenará el byte correspondiente.

3. **Posicionamiento dinámico en el array de destino:**
   - La expresión `i + BYTESPORMUESTRA * numeroMuestrasAcumuladas` asegura que cada nueva muestra se coloque en una posición diferente dentro de `arrayDestino`, evitando la sobrescritura de datos.
   - `BYTESPORMUESTRA` representa el número de bytes que ocupa una única muestra, por lo que `BYTESPORMUESTRA * numeroMuestrasAcumuladas` calcula la posición de inicio del espacio reservado para la siguiente muestra en el array de destino.
4. **Copiado de datos:**
   - Cada byte de `arrayOrigen` se copia en la posición correspondiente de `arrayDestino`. De este modo, los datos de la muestra actual se insertan de manera ordenada en el array de destino, formando un bloque continuo de datos que facilita su procesamiento posterior.

- **Envío de datos:**

```
/**
 * @brief Envía los datos de las muestras acumuladas
 * al servidor a través de UDP.
 *
 * La función combina los datos de las muestras acumuladas
 * en un array de bytes y los envía al servidor.
 * Luego, limpia el buffer de datos y espera un mensaje de parada ("STOP")
 * del servidor para reiniciar la espera del mensaje de inicio ("START").
 *
 * @param byteArrayEnvio Array de bytes que contiene
 * los datos de las muestras acumuladas que se van a enviar.
 * @param byteArray Array de bytes que contiene
 * los datos de la muestra actual que se va a agregar al buffer.
 * @param numeroMuestrasAcumuladas
 * Número de muestras acumuladas en el buffer de envío.
 */
void sendSamples(   byte* byteArrayEnvio,
                    byte* byteArray,
                    int numeroMuestrasAcumuladas){
  // Copiar los datos de la muestra actual al buffer de envío
  copyArrays(byteArrayEnvio, byteArray, contadorMuestras);

  // Iniciar el paquete UDP a la dirección IP del servidor y puerto
  Udp.beginPacket(serverIP, serverPort);

  // Escribir los datos del buffer en el paquete
  Udp.write(byteArrayEnvio, PACKETSIZE);

  // Terminar y enviar el paquete
  Udp.endPacket();

  // Reiniciar el contador de muestras acumuladas
  contadorMuestras = 0;

  // Liberar la memoria del buffer de envío
  free(byteArrayEnvio);

  // Esperar un mensaje de parada del servidor antes de continuar
  waitForStopMessage();
}
```

Código Fuente 5.7: Arduino envío de datos vía UDP

El objetivo de esta función es tomar todas las muestras de datos capturadas y acumuladas en el buffer de envío (`byteArrayEnvio`), y enviarlas como un único paquete al servidor especificado. Después del envío, se libera el buffer de memoria y el sistema espera una señal de parada para evitar la captura y transmisión continuas sin control.

Desglose de la implementación

1. **Copiar la muestra actual al buffer de envío:**
   - La función `copyArrays(byteArrayEnvio, byteArray, contadorMuestras)` se utiliza para copiar los datos de la muestra actual (almacenados en `byteArray`) al buffer de envío (`byteArrayEnvio`).
   - `contadorMuestras` indica cuántas muestras se han acumulado previamente en el buffer. De esta manera, la muestra actual se coloca en la posición correcta dentro del buffer de envío para evitar solapamientos.
2. **Preparar el paquete UDP:**
   - `Udp.beginPacket(serverIP, serverPort)` inicia el paquete UDP, configurando la dirección IP y el puerto del servidor al cual se enviarán los datos.
   - Este paso asegura que los datos se envíen al destino correcto a través de la red.
3. **Escribir los datos en el paquete UDP:**
   - `Udp.write(byteArrayEnvio, PACKETSIZE)` escribe el contenido del buffer de envío en el paquete UDP.
   - `PACKETSIZE` define el tamaño total del paquete que se enviará. Al escribir todos los datos de `byteArrayEnvio` en el paquete, se asegura que las muestras acumuladas se transmitan juntas en un único mensaje.
4. **Enviar el paquete UDP:**
   - `Udp.endPacket()` finaliza la preparación del paquete y lo envía al servidor.
   - Este paso envía físicamente los datos a través de la red al servidor especificado en el paso anterior.
5. **Reiniciar el contador de muestras acumuladas:**
   - `contadorMuestras = 0;` reinicia el contador de muestras, indicando que todas las muestras acumuladas se han enviado correctamente y que el sistema está listo para comenzar a acumular nuevas muestras desde cero.
6. **Liberar la memoria del buffer de envio:**
   - `free(byteArrayEnvio);` libera la memoria asignada al buffer de envío (`byteArrayEnvio`) para evitar fugas de memoria y liberar recursos en el sistema.
   - Este paso es esencial para optimizar el uso de la memoria y garantizar la eficiencia del sistema a largo plazo.
7. **Esperar el mensaje de parada del servidor:**

- `waitForStopMessage();` es una función que espera un mensaje de parada (`"STOP"`) del servidor antes de continuar con el proceso de captura y envío de datos.
- Esto asegura que el sistema no continúe enviando datos innecesariamente, permitiendo un control remoto del proceso de adquisición de datos por parte del servidor.

5.3.5. Mensajes de control

El Arduino también debe manejar los mensajes de control para iniciar y detener la captura de datos:

- **Esperar mensaje de inicio:**

```
/**
 * @brief Espera recibir un mensaje de inicio ("START") desde el servidor.
 *
 * La función se ejecuta en un bucle hasta que se recibe el mensaje "START".
 * La función también extrae el número de segundos que estará activa
 * la adquisición de datos
 */
void waitForStartMessage() {
  while (true) {
    int packetSize = Udp.parsePacket();
    if (packetSize) {
      char packetBuffer[255];
      int len = Udp.read(packetBuffer, 255);
      if (len > 0) {
        packetBuffer[len] = 0;
        char* token = strtok(packetBuffer, ";");
        if (strcmp(token, startMessage) == 0) {
          token = strtok(NULL, ";");
          if (token != NULL) {
            startNumber = atoi(token);
          }
          tiempoInicioCaptura = millis();
          return;
        }
      }
    }
    delay(100);
  }
}
```

Código Fuente 5.8: Arduino esperar mensaje de inicio

El objetivo de esta función es esperar a que el dispositivo reciba el mensaje de inicio `"START"` desde el servidor para comenzar la captura de datos de los sensores. Este mensaje sincroniza la adquisición de datos con la solicitud del servidor, garantizando que el sistema no capture datos innecesarios antes de la orden de inicio.

Desglose de la implementación

1. **Inicialización del bucle de espera:**
   - La función `waitForStartMessage()` se ejecuta en un bucle infinito utilizando `while (true)`, que mantiene al dispositivo en un estado de escucha constante, esperando recibir un paquete UDP.
2. **Verificación de paquetes UDP:**
   - La función `Udp.parsePacket()` se utiliza para comprobar si hay paquetes UDP disponibles en la cola de recepción. Si el valor devuelto es mayor a cero, significa que se ha recibido un paquete.
3. **Lectura de mensaje:**
   - Se almacena el contenido del paquete en el buffer `packetBuffer` de 255 bytes, utilizando `Udp.read(packetBuffer, 255)`. Si se han leído bytes (`len >0`), el carácter nulo (`\0`) se añade al final del buffer para asegurarse de que la cadena esté correctamente terminada.
4. **Análisis del mensaje:**
   - Se extrae el primer token del mensaje recibido con `strtok(packetBuffer, ";")` y se compara con el mensaje esperado (`startMessage`). Si coinciden, significa que se ha recibido la orden de inicio.
5. **Procesamiento de parámetros:**
   - Se extrae un segundo token que contiene un número (opcional). Este número se convierte en un entero utilizando `atoi(token)` y se asigna a `startNumber`, que probablemente indica la duración o la cantidad de muestras que se deben tomar.
6. **Registro del tiempo de inicio:**
   - Se almacena el tiempo de inicio en la variable `tiempoInicioCaptura` utilizando `millis()`, que marca el momento en que comenzó la captura de datos.
7. **Finalización del bucle:**
   - La función se sale del bucle infinito al recibir el mensaje de inicio y retorna al programa principal, permitiendo que el flujo del código continúe con la captura de datos.

- **Esperar mensaje de parada:**

```
/**
 * @brief Espera recibir un mensaje de parada ("STOP") desde el servidor.
 *
 * Si se recibe el mensaje "STOP", se vuelve a llamar a
 * waitForStartMessage() para reiniciar la espera del mensaje de inicio.
 */
void waitForStopMessage() {
  int packetSize = Udp.parsePacket();
  if (packetSize) {
    char packetBuffer[255];
    int len = Udp.read(packetBuffer, 255);
    if (len > 0) {
      packetBuffer[len] = 0;
      if (strcmp(packetBuffer, stopMessage) == 0) {
        waitForStartMessage();
      }
    }
  }
}
```

Código Fuente 5.9: Arduino esperar mensaje de parada

El objetivo de esta función es esperar y detectar el mensaje de parada `"STOP"` enviado por el servidor. Cuando se recibe este mensaje, el sistema detiene la captura de datos y regresa al estado de espera para un nuevo mensaje de inicio.

Desglose de la implementación

1. **Verificación de paquetes UDP:**
   - Al igual que en el código 5.8, la función `Udp.parsePacket()` verifica si hay un paquete UDP disponible en la cola. Si se recibe un paquete, el tamaño de este (`packetSize`) será mayor a cero.
2. **Lectura del mensaje:**
   - Se almacena el contenido del paquete en el buffer `packetBuffer` y se asegura que esté correctamente terminado en nulo para su manejo como una cadena de caracteres.
3. **Análisis del mensaje:**
   - Se compara el contenido del `packetBuffer` con el mensaje esperado `stopMessage ("STOP")` usando `strcmp()`. Si coinciden, se reconoce que se ha recibido la orden de detener la captura.
4. **Retorno al estado de espera:**

- Al recibir el mensaje `"STOP"`, se llama a la función `waitForStartMessage()`, lo que reinicia el proceso de espera por un nuevo mensaje de inicio.

- **Enviar mensaje de broadcast:**

```
/**
 * @brief Envía un mensaje de difusión ("HELLO") a la red
 * y espera una respuesta.
 *
 * La función envía un mensaje de difusión a la dirección IP de broadcast
 * y luego llama a waitForHandshake() para esperar la respuesta.
 */
void sendBroadcastMessage(){
  IPAddress broadcastIP = ~WiFi.subnetMask() | WiFi.gatewayIP();
  Udp.beginPacket(broadcastIP, 5005);
  Udp.write(handshakeMessage);
  Udp.endPacket();
  waitForHandshake();
}
```

Código Fuente 5.10: Arduino enviar mensaje de broadcast

La función `sendBroadcastMessage()` se encarga de enviar un mensaje de difusión (`"HELLO"`) a toda la red, con el objetivo de notificar su presencia a cualquier servidor que esté escuchando. Una vez enviado el mensaje, el dispositivo entra en un estado de espera para recibir una respuesta de handshake del servidor. Este proceso inicial permite que el dispositivo y el servidor establezcan una comunicación antes de comenzar la captura de datos.

Desglose de la implementación

1. **Cálculo de la dirección IP de broadcast:**
   - Se calcula la dirección IP de broadcast utilizando la máscara de subred (`WiFi.subnetMask()`) y la puerta de enlace (`WiFi.gatewayIP()`).
   - `~WiFi.subnetMask()`: Invierte todos los bits de la máscara de subred. Por ejemplo, si la máscara de subred es `255.255.255.0` (en binario `11111111.11111111.11111111.00000000`), la operación `~` cambia esto a `00000000.00000000.00000000.11111111`.
   - `WiFi.gatewayIP()`: Recupera la dirección IP de la puerta de enlace. Al aplicar un OR bit a bit (`|`) entre la dirección invertida de la subred y la puerta de enlace, se obtiene la dirección IP de broadcast. Esta dirección se utiliza para enviar mensajes a todos los dispositivos de la red.
2. **Inicio del paquete de broadcast:**

- `Udp.beginPacket(broadcastIP, 5005);` prepara un paquete UDP para ser enviado a la dirección de broadcast calculada (`broadcastIP`) en el puerto `5005`.
- El puerto `5005` se utiliza como un canal dedicado para el envío y recepción de mensajes de control en esta aplicación, asegurando que el mensaje sea recibido por todos los dispositivos que estén escuchando en ese puerto.

3. **Escritura del mensaje de handshake:**
   - `Udp.write(handshakeMessage);` escribe el mensaje de saludo `"HELLO"` en el paquete UDP. Este mensaje sirve para anunciar la presencia del dispositivo a la red y para iniciar el proceso de handshake con el servidor.
4. **Envío del paquete de broadcast:**
   - `Udp.endPacket();` finaliza y envía el paquete de broadcast con el mensaje `"HELLO"`. Este paquete es transmitido a todos los dispositivos de la red que estén escuchando en el puerto `5005`.
5. **Esperar respuesta de handshake:**
   - Después de enviar el mensaje de broadcast, la función llama a `waitForHandshake()` para esperar una respuesta del servidor.
   - `waitForHandshake()` permanece en un bucle de escucha hasta recibir un mensaje de saludo de vuelta desde el servidor, indicando que el servidor ha reconocido el mensaje de broadcast y está listo para establecer comunicación.

- **Esperar mensaje de broadcast:**

```
/**
 * @brief Espera recibir un mensaje de saludo ("HELLO") desde el servidor.
 *
 * La función se ejecuta en un bucle
 * hasta que se recibe el mensaje "HELLO".
 */
void waitForHandshake(){
  while(1){
    int packetSize = Udp.parsePacket();
    if(packetSize){
      char packetBuffer[255];
      int len = Udp.read(packetBuffer, 255);
      if(len > 0){
        packetBuffer[len] = 0;
        if(strcmp(packetBuffer, handshakeMessage) == 0){
          serverIP = Udp.remoteIP();
          serverPort = Udp.remotePort();
          return;
        }
      }
    }
    delay(100);
  }
}
```

Código Fuente 5.11: Arduino esperar mensaje de handshake

Esperar hasta recibir un mensaje de saludo (`"HELLO"`) desde el servidor. Este mensaje de handshake permite que el dispositivo Arduino identifique la dirección IP y el puerto del servidor con el que se comunicará durante la sesión de captura de datos. La función asegura que tanto el dispositivo como el servidor estén listos y sincronizados para intercambiar datos.

Desglose de la implementación

1. **Bucle de espera:**
   - La función comienza con un bucle `while(1)`, que mantiene al Arduino en un estado de escucha constante para recibir el mensaje de saludo. Este bucle infinito garantiza que el dispositivo no salga del estado de espera hasta que se reciba el mensaje `"HELLO"`.
2. **Verificación de paquetes UDP:**
   - `Udp.parsePacket()` verifica si hay paquetes UDP disponibles en la cola de recepción. Si el valor devuelto (`packetSize`) es mayor que cero, significa que hay un

paquete entrante que debe ser procesado. Este paso inicial filtra las entradas válidas que podrían contener el mensaje de handshake.

3. **Lectura del contenido del paquete:**
   - Se utiliza `Udp.read(packetBuffer, 255)` para leer el contenido del paquete y almacenarlo en el buffer `packetBuffer`. La variable `len` almacena el número de bytes leídos del paquete. Si `len` es mayor que cero, significa que el paquete contiene datos, y se añade un carácter nulo (`\0`) al final del buffer para asegurarse de que la cadena esté correctamente terminada y pueda procesarse como texto.
4. **Comprobación del mensaje de handshake:**
   - `strcmp(packetBuffer, handshakeMessage)` compara el contenido del buffer recibido con el mensaje esperado `"HELLO"`. Si ambos coinciden, se confirma que el servidor ha enviado el mensaje de saludo correcto. Esta verificación es fundamental para asegurarse de que el dispositivo solo reaccione al mensaje esperado y no a otros posibles datos recibidos en la red.
5. **Registro de la dirección IP y el puerto del servidor:**
   - `serverIP = Udp.remoteIP();` guarda la dirección IP del servidor que envió el mensaje en la variable `serverIP`.
   - `serverPort = Udp.remotePort();` almacena el puerto desde el cual el servidor envió el mensaje en la variable `serverPort`.
   - Estos datos se utilizarán posteriormente para enviar paquetes de datos de sensores al servidor correcto, garantizando que la comunicación sea precisa y dirigida.
6. **Salida del bucle y retorno:**
   - La función termina con `return;`, lo que provoca que se salga del bucle y se permita que el programa principal continúe su ejecución. Esto indica que el handshake se ha completado con éxito y que el dispositivo está listo para recibir comandos adicionales o iniciar la captura de datos.
7. **Retardo en el bucle de espera:**
   - `delay(100);` introduce un retardo de 100 milisegundos en cada iteración del bucle si no se ha recibido un paquete válido. Esto reduce el consumo de recursos y evita que el bucle se ejecute a una velocidad innecesariamente alta, optimizando el rendimiento del dispositivo durante la espera.

5.4. Implementación del servidor Flask

En esta sección se detalla la implementación del servidor Flask, que actúa como la pieza central del sistema, permitiendo la recepción de datos desde la placa Arduino Nano IoT 33, su procesamiento y la interacción con el usuario a través de un panel web. La arquitectura del servidor Flask se complementa con

un servidor UDP para la comunicación directa con el Arduino, asegurando la captura y el almacenamiento de los datos obtenidos por los sensores MEMS.

5.4.1. Recepción de datos

El proceso de recepción de datos comienza configurando un socket UDP que permite la comunicación entre el servidor y la placa Arduino. La función principal para esta tarea es `setup_socket()`, que escucha las transmisiones UDP desde la placa y espera a recibir un mensaje de `"HELLO"` para confirmar la conexión. Esta función asegura que el servidor esté correctamente configurado para recibir los datos enviados por el Arduino y establece el punto de inicio de la transmisión de datos.

```
# Función para configurar el socket UDP
def setup_socket():
    global server_socket, UDP_IP, UDP_PORT,
    BUFFER_SIZE, ARDUINO_IP, ARDUINO_PORT
    # Configuración de las variables globales relacionadas
    # con la conexión UDP
    UDP_IP = ''           # Dirección IP local para
                          # escuchar en todas las interfaces disponibles
    UDP_PORT = 5005       # Puerto UDP donde escuchará el servidor
    BUFFER_SIZE = 560     # Tamaño del buffer para recibir datos
                          # (20 muestras de 7 floats)
    # Dirección IP y puerto del Arduino (se inicializan vacíos)
    ARDUINO_IP = ''
    ARDUINO_PORT = 0
    # Creación de un socket UDP
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # Configuración para permitir broadcast en el socket
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    # Intento de obtener la dirección IP del servidor local
    try:
        UDP_IP = socket.gethostbyname(socket.gethostname())
    except socket.gaierror:
        print("Error: No se puede obtener la dirección IP.")
        exit()
    # Vinculación del socket al puerto especificado
    server_socket.bind(("", UDP_PORT))
    # Bucle principal para esperar la conexión y recibir datos
    while True:
        # Recibir datos y dirección del cliente (Arduino)
        data, addr = server_socket.recvfrom(BUFFER_SIZE)
        message = data.decode()
        # Si el mensaje recibido es "HELLO",
        # se establece la dirección IP y puerto del Arduino
        if message == "HELLO":
            ARDUINO_IP = addr[0]
            ARDUINO_PORT = addr[1]
            response = "HELLO"
            # Enviar respuesta "HELLO" de vuelta al Arduino
            server_socket.sendto(response.encode(), addr)
            break  # Salir del bucle una vez establecida la conexión
    # Mensaje indicando que el servidor UDP está iniciado y esperando datos
print("Servidor UDP iniciado. Esperando datos...")
```

Código Fuente 5.12: Flask configuración inicial del socket UDP

Desglose de la implementación

1. **Declaración de variables globales:**

```
global server_socket, UDP_IP, UDP_PORT,
    BUFFER_SIZE, ARDUINO_IP, ARDUINO_PORT
```

   - `server_socket`: El socket UDP que se utilizará para recibir datos.
   - `UDP_IP`: La dirección IP local en la que el servidor escuchará los datos.
   - `UDP_PORT`: El puerto UDP en el que el servidor escuchará (5005 en este caso).
   - `BUFFER_SIZE`: El tamaño del buffer de datos a recibir, definido como 560 bytes, que corresponde a 20 muestras de 7 floats.
   - `ARDUINO_IP` y `ARDUINO_PORT` : Se inicializan vacíos y se rellenan con la dirección IP y puerto del Arduino después del "handshake".

2. **Configuración de la conexión UDP:**

```
# Dirección IP local para escuchar en todas las interfaces
# disponibles
UDP_IP = ''
# Puerto UDP donde escuchará el servidor
UDP_PORT = 5005
# Tamaño del buffer para recibir datos (20 muestras de 7 floats)
BUFFER_SIZE = 560
```

   - `UDP_IP`: Se establece inicialmente como una cadena vacía, lo que indica que el servidor escuchará en todas las interfaces disponibles en el sistema.
   - `UDP_PORT`: Especifica el puerto 5005 donde el servidor escuchará las conexiones.
   - `BUFFER_SIZE`: Define el tamaño del buffer para recibir datos, lo que corresponde al tamaño de los paquetes enviados por el Arduino (20 muestras de 7 floats).

3. **Creación del socket UDP:**

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

   - `server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`: Crea un socket UDP utilizando el protocolo IPv4 (`AF_INET`) y el protocolo de transporte UDP (`SOCK_DGRAM`). UDP es ideal para la transmisión rápida de datos en tiempo real.

4. **Permitir el uso de broadcast en el socket:**

```
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
```

   - Permite el uso de mensajes de broadcast, lo que significa que los mensajes pueden enviarse a todos los dispositivos en la red local que estén escuchando en el puerto correspondiente.

5. **Obtener la dirección IP local del servidor:**

```
try:
    UDP_IP = socket.gethostbyname(socket.gethostname())
except socket.gaierror:
    print("Error: No se puede obtener la dirección IP.")
    exit()
```

   - `socket.gethostbyname(socket.gethostname())`: Intenta obtener la dirección IP del servidor a partir de su nombre de host. En caso de error (`socket.gaierror`), se muestra un mensaje y se cierra el programa.

6. **Vinculación del socket:**

```
server_socket.bind(("", UDP_PORT))
```

   - Vincula el socket a todas las interfaces disponibles (`''`) y al puerto UDP definido (`5005`), lo que permite al servidor escuchar en este puerto.

7. **Bucle principal para recibir datos:**

```
while True:
    data, addr = server_socket.recvfrom(BUFFER_SIZE)
    message = data.decode()
```

   - `server_socket.recvfrom(BUFFER_SIZE)`: Espera y recibe datos del cliente (Arduino) con un tamaño máximo de `BUFFER_SIZE`. `addr` contiene la dirección del cliente.
   - `data.decode()`: Decodifica los datos recibidos en formato de texto.

8. **Proceso de handshake:**

```
if message == "HELLO":
    ARDUINO_IP = addr[0]
    ARDUINO_PORT = addr[1]
    response = "HELLO"
    server_socket.sendto(response.encode(), addr)
    break
```

- Si el mensaje recibido es `"HELLO"`, se establece la dirección IP y el puerto del Arduino, lo que completa el handshake inicial. El servidor responde con otro mensaje `"HELLO"` para confirmar la conexión.
- El bucle se rompe tras este intercambio para proceder con la siguiente fase de la conexión.

9. **Mensaje de confirmación:**

```
print("Servidor UDP iniciado. Esperando datos...")
```

- Un mensaje que confirma que el servidor UDP está en funcionamiento y listo para recibir más datos.

5.4.2. Procesamiento de los datos recibidos:

Una vez que el servidor recibe los datos binarios enviados por el Arduino, estos deben ser procesados. Los datos recibidos contienen lecturas de los sensores en formato binario, por lo que es necesario desempaquetarlos y convertirlos a `floats` para su posterior análisis y almacenamiento.

```
# Función para mostrar resultados parseados y guardarlos en CSV
def show_results(data):
# Suponiendo 20 muestras de 7 floats cada una (formato little-endian)
    fmt = '<140f'
# Desempaquetar los datos binarios según el formato especificado
    samples = struct.unpack(fmt, data)
    for i in range(20):
          # Extraer datos para cada muestra (7 floats)
        sample_data = samples[i*7:(i+1)*7]
        accelerometer_x, accelerometer_y, accelerometer_z,
        gyro_x, gyro_y, gyro_z,
        timestamp = sample_data

        # Escribir los datos en el archivo CSV 'data.csv'
        with open('data.csv', 'a', newline='') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow([
                timestamp,
                accelerometer_x, accelerometer_y, accelerometer_z,
                gyro_x, gyro_y, gyro_z])
```

Código Fuente 5.13: Flask procesamiento de los datos recibidos

Desglose de la implementación

1. **Función `show_results(data):`**

   ```
   def show_results(data):
   ```

   - Esta función recibe un argumento `data`, que se espera sea un conjunto de datos binarios empaquetados enviados por el Arduino. Estos datos contienen mediciones del acelerómetro, giroscopio y un sello de tiempo para cada muestra.

2. **Especificación del formato de los datos binarios:**

   ```
   fmt = '<140f'
   ```

   - `<`: El carácter < indica que los datos están en formato **little-endian**, lo cual es un estándar en la transmisión de datos en muchas plataformas de hardware.

- `140f`: Se espera que los datos contengan **140 floats**. Como cada muestra contiene 7 floats, y hay 20 muestras (7 floats × 20 muestras = 140 floats en total), se desempaquetan los datos usando este formato.

3. **Desempaquetado de los datos binarios:**

```
samples = struct.unpack(fmt, data)
```

- `struct.unpack(fmt, data)`: La función `unpack` toma los datos binarios (en este caso, 140 floats en formato little-endian) y los convierte en una tupla de floats legibles. Los datos desempaquetados se almacenan en la variable `samples`.

4. **Extracción de las muestras:**

```
for i in range(20):
    sample_data = samples[i*7:(i+1)*7]
    accelerometer_x, accelerometer_y, accelerometer_z,
    gyro_x, gyro_y, gyro_z, timestamp = sample_data
```

- `for i in range(20)`: Se itera 20 veces, una vez por cada muestra recibida.
- `sample_data = samples[i*7:(i+1)*7]`: En cada iteración, se extraen 7 floats de la lista `samples`, que corresponden a los valores de aceleración en los ejes X, Y y Z, velocidad angular (giroscopio) en los ejes X, Y y Z, y el sello de tiempo.
- Los valores se asignan a las variables: `accelerometer_x`, `accelerometer_y`, `accelerometer_z`, `gyro_x`, `gyro_y`, `gyro_z`, y `timestamp`.

5. **Escritura de los datos en un archivo CSV:**

```
with open('data.csv', 'a', newline='') as csvfile:
    csvwriter = csv.writer(csvfile)
    csvwriter.writerow([
        timestamp,
        accelerometer_x, accelerometer_y, accelerometer_z,
        gyro_x, gyro_y, gyro_z])
```

- `with open('data.csv', 'a', newline='') as csvfile`: Abre (o crea si no existe) el archivo `data.csv` en modo append (`'a'`), lo que significa que los nuevos datos se añadirán al final del archivo sin sobrescribir los datos existentes. `newline=''` asegura que no haya líneas en blanco entre filas.

- `csv.writer(csvfile)`: Crea un objeto escritor CSV que permitirá escribir filas de datos en el archivo.
- `csvwriter.writerow([...])`: Escribe una fila de datos en el archivo CSV, que contiene el `timestamp`, las lecturas del acelerómetro (`accelerometer_x`, `accelerometer_y`, `accelerometer_z`,) y las lecturas del giroscopio (`gyro_x`, `gyro_y`, `gyro_z`).

5.4.3. Comandos desde el panel web

El sistema permite la interacción del usuario a través de un panel web gestionado por el servidor Flask. El panel ofrece la posibilidad de **iniciar** o **detener** la captura de datos, así como de realizar pruebas para verificar el estado de los sensores. Estos comandos se manejan mediante solicitudes HTTP enviadas desde la interfaz web a través de diferentes rutas definidas en Flask.

```
@app.route('/command')
def command():
    # Obtener el valor del parámetro 'cmd' de la solicitud
    cmd = request.args.get('cmd')
    # Obtener el valor de 'time' como entero,
    # valor por defecto 10 si no está presente
    time = request.args.get('time', default=10, type=int)

    if cmd == 'start':
        # Llamar a la función send_start_message() con el número 0
        send_start_message(0)
    elif cmd == 'start_temp':
        # Llamar a send_start_message() con el valor de 'time'
        send_start_message(time)
    elif cmd == 'stop':
    # Llamar a la función send_stop_message() para detener la operación
        send_stop_message()
    # Devolver una respuesta JSON indicando el comando recibido
    return jsonify({'message': 'Command received: ' + cmd})
```

Código Fuente 5.14: Flask comandos desde el panel web

Esta ruta se encarga de recibir comandos desde el panel web. Dependiendo del comando (`cmd`) recibido, se inicia o detiene la captura de datos, como en:

- `@app.route('/command')`: Define una ruta en el servidor Flask que responde a solicitudes enviadas por el panel web.
- `cmd = request.args.get('cmd')`: Obtiene el valor del parámetro `cmd` de la solicitud HTTP. Este valor puede ser `'start'`, `'start_temp'`, o `'stop'`.

- `time = request.args.get('time', default=10, type=int)`: Obtiene el parámetro time (si existe) como un entero. Si no se proporciona, el valor por defecto es 10.
    - Comandos:
        - `'start'`: Llama a `send_start_message(0)`, que indica al Arduino que comience la captura continua de datos.
        - `'start_temp'`: Inicia la captura temporizada, enviando el tiempo definido al Arduino.
        - `'stop'`: Llama a `send_stop_message()`, deteniendo la captura de datos.
    - `return jsonify('message': 'Command received: ' + cmd)`: Devuelve una respuesta JSON indicando el comando recibido.

Prueba de sensores

Otra funcionalidad disponible en el panel web es la posibilidad de realizar una prueba de sensores de 10 segundos, donde el servidor recopila datos y los muestra al usuario para verificar el correcto funcionamiento de los sensores.

```
def test_sensors():
 if ARDUINO_IP != '':
     # Enviar mensaje de inicio con el número 10
     start_message = f"START;10".encode()
     server_socket.sendto(start_message, (ARDUINO_IP, ARDUINO_PORT))
     # Establecer el tiempo de finalización (10 segundos desde ahora)
     end_time = time.time() + 10
     # Intervalo de tiempo entre cada muestra
     # (aproximadamente 1/25 segundos)
     interval = 1 / 50
     # Lista para almacenar los datos de las muestras
     data_points = []
         while time.time() < end_time:
         # Recibir datos del socket
         data, addr = server_socket.recvfrom(BUFFER_SIZE)
         # Desempaquetar los datos recibidos
         # (suponiendo que son 140 floats
        en formato little-endian)
         fmt = '<140f'
         samples = struct.unpack(fmt, data)
         # Tomar solo la primera muestra de 7 floats para simplificar
         sample_data = samples[:7]
         # Agregar los datos de la muestra a la lista de puntos de datos
         data_points.append(sample_data)
         # Esperar el intervalo antes de la próxima muestra
         time.sleep(interval)
 # Devolver la lista de puntos de datos recolectados durante la prueba
 return data_points
```

Código Fuente 5.15: Flask prueba de sensores

Desglose de la implementación

1. **Verificación de la conexión con el Arduino:**

   ```
   if ARDUINO_IP != '':
   ```

   - Esta condición verifica si la dirección IP del Arduino (`ARDUINO_IP`) ha sido establecida. Si `ARDUINO_IP` es una cadena vacía (”), significa que el Arduino no ha sido detectado y no se puede proceder con la prueba.

2. **Enviar mensaje de inicio a Arduino:**

```
start_message = f"START;10".encode()
server_socket.sendto(start_message, (ARDUINO_IP, ARDUINO_PORT))
```

   - `start_message`: El mensaje `START;10` se envía al Arduino indicando que debe iniciar la captura de datos por 10 segundos. El número `10` en el mensaje indica la duración de la captura.
   - `start_message.encode()`: Convierte el mensaje en una cadena de bytes, que es el formato necesario para enviarlo por la red.
   - `server_socket.sendto()`: Envía el mensaje de inicio a la dirección IP (`ARDUINO_IP`) y puerto (`ARDUINO_PORT`) del Arduino.

3. **Establecer el tiempo de finalización:**

```
end_time = time.time() + 10
```

   - `time.time()`: Devuelve el tiempo actual en segundos desde el "epoch" (1 de enero de 1970). Al sumarle `10`, se establece que la prueba finalizará exactamente 10 segundos después del momento en que se ejecuta esta línea de código.

4. **Configurar el intervalo entre muestras:**

```
interval = 1 / 50
```

   - El intervalo entre cada muestra se fija en aproximadamente **1/50 segundos**, lo que corresponde a una frecuencia de captura de 50 muestras por segundo.

5. **Inicializar la lista de puntos de datos:**

```
data_points = []
```

   - `data_points`: Es una lista vacía que se utilizará para almacenar los datos capturados durante los 10 segundos de la prueba.

6. **Bucle para capturar datos durante 10 segundos:**

```
while time.time() < end_time:
    data, addr = server_socket.recvfrom(BUFFER_SIZE)
    ...
    time.sleep(interval)
```

- `while time.time() <end_time`: El bucle se ejecuta hasta que el tiempo actual (`time.time()`) sea mayor o igual a `end_time` (10 segundos después de haber iniciado la prueba).
- `server_socket.recvfrom(BUFFER_SIZE)`: El servidor escucha y recibe datos del Arduino. `BUFFER_SIZE` define el tamaño máximo del paquete de datos que puede recibir, que en este caso es suficiente para 20 muestras (560 bytes).
- `time.sleep(interval)`: El programa espera el intervalo definido antes de solicitar la siguiente muestra de datos.

7. **Desempaquetar los datos recibidos:**

```
fmt = '<140f'
samples = struct.unpack(fmt, data)
```

- `fmt = '<140f'`: Define el formato para desempaquetar los datos recibidos. Se espera un total de 140 floats (20 muestras × 7 floats) en formato little-endian.
- Convierte los datos binarios recibidos en una lista de 140 floats.

8. **Extraer y almacenar solo la primera muestra:**

```
sample_data = samples[:7]
data_points.append(sample_data)
```

- `samples[:7]`: Se extraen los primeros 7 floats de los datos desempaquetados, que corresponden a la primera muestra de la serie (acelerómetro en X, Y, Z; giroscopio en X, Y, Z; y el timestamp).
- `data_points.append(sample_data)`: La primera muestra extraída se agrega a la lista `data_points`.

9. **Esperar antes de la próxima muestra:**

```
time.sleep(interval)
```

- El programa espera el tiempo definido por interval (`1/50 segundos`) antes de continuar con la próxima iteración y captura de datos.

10. **Devolver los datos recolectados:**

```
return data_points
```

- Al finalizar los 10 segundos de captura, la función devuelve la lista `data_points`, que contiene todas las muestras capturadas durante la prueba.

Verificación y descarga de datos

El servidor Flask también incluye rutas que permiten al usuario verificar los datos capturados y descargar el archivo CSV. Estas rutas gestionan la lectura del archivo data.csv y su entrega al usuario, ya sea en formato JSON para su visualización o como un archivo descargable.

```
@app.route('/check_data')
def check_sensor_data():
    data = []

    # Verificar si el archivo existe
    if not os.path.exists('data.csv'):
        abort(404, description="Archivo 'data.csv' no encontrado")

    # Leer data.csv y guardar en data
    with open('data.csv', 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            data.append([float(value) for value in row])

    return jsonify(data)
```

Código Fuente 5.16: Flask verificación de datos

```python
@app.route('/download_csv')
def download_csv():
    # Leer el archivo CSV existente
    filename = request.args.get('filename')
    input_file = 'data.csv'
    processed_data = []

    with open(input_file, mode='r', newline='') as file:
        reader = csv.reader(file)
        headers = next(reader)  # Leer las cabeceras
        for row in reader:
            # Aquí puedes procesar cada fila según sea necesario
            processed_data.append(row)

    # Crear el archivo CSV en memoria con los datos procesados
    si = io.StringIO()
    cw = csv.writer(si)
    cw.writerow(headers)  # Escribir las cabeceras
    cw.writerows(processed_data)  # Escribir los datos procesados
    output = io.BytesIO()
    output.write(si.getvalue().encode('utf-8'))
    output.seek(0)

    # Eliminamos el archivo.csv
    try:
        os.remove(input_file)
        print(f"File {input_file} deleted successfully.")
    except FileNotFoundError:
        print(f"File {input_file} not found.")
    except PermissionError:
        print(f"Permission denied: Unable to delete {input_file}.")
    except Exception as e:
        print(f"Error occurred while deleting file {input_file}: {e}")

    # Enviar el archivo al cliente
    return send_file(output, mimetype='text/csv',
    as_attachment=True, download_name=f'{filename}.csv')
```

Código Fuente 5.17: Flask descarga de datos

Desglose de la implementación:

- **Función `check_sensor_data()`:**

  1. **Ruta y función para verificar datos:**

     ```
     @app.route('/check_data')
     def check_sensor_data():
     ```

     - `@app.route('/check_data')`: Esta ruta define un endpoint para verificar los datos capturados. Cuando un cliente realiza una solicitud a esta ruta, se llama a la función `check_sensor_data()`.
     - `check_sensor_data()`: Esta función comprueba si el archivo `data.csv` existe y, si es así, lee su contenido, lo convierte a una lista de flotantes, y lo devuelve en formato JSON.

  2. **Verificar si el archivo CSV existe:**

     ```
     if not os.path.exists('data.csv'):
         abort(404, description="Archivo 'data.csv' no encontrado")
     ```

     - `os.path.exists('data.csv')`:Comprueba si el archivo `data.csv` existe en el sistema.
     - `abort(404, description="Archivo 'data.csv' no encontrado")`: Si el archivo no existe, se genera un error HTTP 404 con el mensaje personalizado.

  3. **Leer y procesar el archivo CSV:**

     ```
     with open('data.csv', 'r') as file:
         csv_reader = csv.reader(file)
         for row in csv_reader:
             data.append([float(value) for value in row])
     ```

     - `open('data.csv', 'r')`: Abre el archivo `data.csv` en modo lectura.
     - `csv.reader(file)`: Lee el contenido del archivo utilizando el lector de CSV.
     - `for row in csv_reader`: Itera sobre cada fila en el archivo CSV.
     - `data.append([float(value) for value in row])`: Convierte cada valor de la fila a un flotante y lo agrega a la lista `data`.

  4. **Devolver los datos en formato JSON:**

     ```
     return jsonify(data)
     ```

     - `jsonify(data)`: Convierte la lista `data` en formato JSON y la devuelve como respuesta al cliente.

- **Función `download_csv()`**

1. **Ruta y función para descargar el archivo CSV**

```
@app.route('/download_csv')
def download_csv():
```

   - @app.route(`'/download_csv'`): Esta ruta define el endpoint para descargar el archivo CSV. Cuando un cliente realiza una solicitud a esta ruta, se llama a la función `download_csv()`.
   - `download_csv()`: Esta función permite descargar el archivo `data.csv`, procesarlo, y luego eliminarlo del sistema una vez descargado.

2. **Obtener el nombre del archivo y leer el archivo CSV:**

```
filename = request.args.get('filename')
input_file = 'data.csv'
```

   - `filename = request.args.get('filename')`: Obtiene el nombre del archivo que el usuario ha solicitado descargar a través de los parámetros de la URL.
   - `input_file = 'data.csv'`: Define el archivo data.csv como el archivo que será procesado y descargado.

3. **Procesar el archivo CSV:**

```
with open(input_file, mode='r', newline='') as file:
    reader = csv.reader(file)
    headers = next(reader)  # Leer las cabeceras
    for row in reader:
        processed_data.append(row)
```

   - `with open(input_file, mode='r', newline='') as file`: Abre el archivo `data.csv` en modo lectura.
   - `headers = next(reader)`: Lee las cabeceras del archivo CSV.
   - `processed_data.append(row)`: Agrega cada fila de datos a la lista `processed_data`.

4. **Crear un archivo CSV en memoria:**

```
si = io.StringIO()
cw = csv.writer(si)
cw.writerow(headers)  # Escribir las cabeceras
cw.writerows(processed_data)  # Escribir los datos procesados
```

- `si = io.StringIO()`: Crea un buffer en memoria para almacenar el contenido del archivo CSV.
- `cw.writerow(headers)`: Escribe las cabeceras en el archivo CSV.
- `cw.writerows(processed_data)`: Escribe las filas procesadas en el archivo CSV.

5. **Escribir el archivo en un buffer de salida:**

```
output = io.BytesIO()
output.write(si.getvalue().encode('utf-8'))
output.seek(0)
```

- `output = io.BytesIO()`: Crea un buffer de bytes en memoria.
- `output.write(si.getvalue().encode('utf-8'))`: Escribe el contenido del archivo CSV en el buffer de bytes.
- `output.seek(0)`: Coloca el cursor al principio del buffer para que se pueda leer desde el inicio al enviar el archivo al cliente.

6. **Eliminar el archivo CSV del sistema:**

```
try:
    os.remove(input_file)
    print(f"File {input_file} deleted successfully.")
except FileNotFoundError:
    print(f"File {input_file} not found.")
except PermissionError:
    print(f"Permission denied: Unable to delete {input_file}.")
except Exception as e:
    print(f"Error occurred while deleting file {input_file}: {e}")
```

- `os.remove(input_file)`: Intenta eliminar el archivo data.csv del sistema.
- Los bloques `except` manejan diferentes errores como `FileNotFoundError`, `PermissionError`, y otros errores imprevistos.

7. **Enviar el archivo CSV al cliente:**

```
return send_file(   output, mimetype='text/csv', as_attachment=True,
                    download_name=f'{filename}.csv')
```

- `send_file(output, mimetype='text/csv', as_attachment=True, download_name=f'filename.csv')`: Envía el archivo CSV procesado al cliente como un archivo adjunto con el nombre proporcionado en `filename`.

5.5. Interfaz de usuario (Panel Web)

La interfaz de usuario implementada en este proyecto permite al usuario interactuar con el sistema para iniciar la captura de datos, detenerla, realizar pruebas de sensores, y visualizar los datos en tiempo real mediante gráficos interactivos. Esta interfaz ha sido desarrollada utilizando **HTML5**, **CSS** y **JavaScript**, lo que garantiza una experiencia amigable y eficaz para el usuario.

5.5.1. Componentes del panel web

El archivo index.html contiene la estructura principal del panel web. A continuación, se detallan los componentes principales del panel, junto con fragmentos de código que ilustran su funcionamiento.

Botones de control

El panel web cuenta con varios botones que permiten controlar la captura de datos y la interacción con el sistema:

```
<div class="buttons-area">
    <div class="button-container">
        <div class="button-wrapper">
            <button id="startButton"
                    onclick="sendCommand('start')">
                    COMENZAR
            </button>
        </div>
        <div class="temp-container button-wrapper tiempo">
            <button id="startButtonTemp"
                    onclick="sendCommand('start_temp')">
                    COMENZAR TEMPORIZADO
            </button>
            <label for="time">N° seg:</label>
            <input  type="number"
                    id="time"
                    name="time"
                    min="1"
                    value="10">
        </div>
        <div class="button-wrapper">
            <button id="stopButton"
                    onclick="sendCommand('stop')">
                    PARAR
            </button>
        </div>
    </div>
</div>
```

Código Fuente 5.18: Web panel botones de control

- **Botón "COMENZAR":** Inicia la captura continua de datos desde el Arduino. Cuando el usuario presiona este botón, se envía un comando al servidor Flask para comenzar la transmisión de datos.
- **Botón "COMENZAR TEMPORIZADO":** Inicia una captura de datos durante un tiempo especificado. El usuario puede seleccionar el tiempo en segundos en el campo input, y el sistema detendrá automáticamente la captura una vez que finalice el tiempo.
- **Botón "PARAR":** Detiene la captura de datos en cualquier momento, ya sea una captura continua o temporizada.

Temporizador

El sistema incluye un temporizador que muestra el tiempo restante cuando se realiza una captura temporizada:

```
<h2>Temporizador: <span id="minutes">
                        0
                    </span> minutos
                    <span id="seconds">
                        0
                    </span> segundos
</h2>

function startCountdown(duration) {
            clearInterval(countdownInterval);
            let totalSeconds = duration;
            let minutes = Math.floor(totalSeconds / 60);
            let seconds = totalSeconds % 60;

            document.getElementById('minutes').innerText = minutes;
            document.getElementById('seconds').innerText = seconds;

            countdownInterval = setInterval(() => {
                totalSeconds--;

                minutes = Math.floor(totalSeconds / 60);
                seconds = totalSeconds % 60;

                document.getElementById('minutes').innerText = minutes;
                document.getElementById('seconds').innerText = seconds;

                if (totalSeconds <= 0) {
                    clearInterval(countdownInterval);
                    enableButtons();
                }
            }, 1000);
        }
```

Código Fuente 5.19: Web panel temporizador

Este temporizador es gestionado mediante una función JavaScript que actualiza el tiempo cada segundo durante la captura temporizada. Este código permite que el temporizador se actualice dinámicamente mientras el usuario observa el proceso de captura.

Visualización de datos en gráficos

Los datos recogidos por los sensores se muestran en tiempo real en dos gráficos, uno para las lecturas del **acelerómetro** y otro para el **giroscopio**. Estos gráficos son generados con la biblioteca **Chart.js**.

```
<div class="chart-wrapper">
    <canvas id="chart1"></canvas>
    <div id="chart1-buttons">
        <button onclick="toggleDatasetVisibility(chart1, 0)">
                Toggle Accelerometro X
        </button>
        <button onclick="toggleDatasetVisibility(chart1, 1)">
                Toggle Accelerometro Y
        </button>
        <button onclick="toggleDatasetVisibility(chart1, 2)">
                Toggle Accelerometro Z
        </button>
    </div>
</div>
<div class="chart-wrapper">
    <canvas id="chart2"></canvas>
    <div id="chart2-buttons">
        <button onclick="toggleDatasetVisibility(chart2, 0)">
                Toggle Giroscopio X
        </button>
        <button onclick="toggleDatasetVisibility(chart2, 1)">
                Toggle Giroscopio Y
        </button>
        <button onclick="toggleDatasetVisibility(chart2, 2)">
                Toggle Giroscopio Z
        </button>
    </div>
</div>
```

```
const chart1 = new Chart(ctx1, {
    type: 'line',
    data: {
        labels: [],
        datasets: colorsAccel.map((color, index) => ({
            label: `Acelerómetro ${index}`, data: [], borderColor: color
        }))
    }
});
const chart2 = new Chart(ctx2, {
    type: 'line',
    data: {
        labels: [],
        datasets: colorsGyros.map((color, index) => ({
            label: `Giroscopio ${index}`, data: [], borderColor: color
        }))
    }
});
```

Código Fuente 5.20: Web panel visualización de datos en gráficos

- **Gráfico acelerómetro:** Representa los datos de aceleración en los ejes X, Y y Z.
- **Gráfico giroscopio:** Muestra las lecturas del giroscopio para los tres ejes.
- **Botones de alternancia:** Permiten al usuario habilitar o deshabilitar la visibilidad de los conjuntos de datos específicos (por ejemplo, desactivar la visualización de aceleración en el eje X).

5.6. Alamcenamiento de datos

En este proyecto, el almacenamiento de los datos capturados es una funcionalidad esencial, ya que permite al usuario guardar las lecturas generadas por los sensores MEMS para su análisis posterior. El sistema almacena los datos en archivos CSV (Comma Separated Values), un formato ampliamente utilizado para el manejo de datos estructurados, fácilmente accesible por herramientas como Excel, Python o R.

5.6.1. Formato de almacenamiento

Los datos de cada sesión de captura se almacenan en un archivo CSV. Este archivo contiene múltiples filas, cada una correspondiente a una muestra de datos obtenida durante la captura, donde cada columna representa una variable medida por los sensores.

Cada archivo CSV generado contiene las siguientes columnas:

- **TimeStamp:** Marca de tiempo en milisegundos en que se capturó la muestra.
- **Acelerómetro X, Y, Z:** Valores de aceleración en los tres ejes (X, Y, Z).
- **Giroscopio X, Y, Z:** Valores de la velocidad angular en los tres ejes (X, Y, Z).

Este formato estructurado facilita la organización y análisis de los datos, ya que cada línea de datos representa una lectura completa de los sensores.ç

5.6.2. Proceso de almacenamiento

El proceso de almacenamiento se divide en tres pasos principales: captura de datos, procesamiento en el servidor y guardado manual por parte del usuario.

1. **Captura de datos**

   El primer paso es la captura de datos por parte de los sensores MEMS en la raqueta, transmitidos a través de la placa Arduino Nano IoT 33 mediante UDP al servidor Flask. Este servidor recibe los datos en formato binario y los procesa para convertirlos en valores flotantes utilizables.

2. **Procesamiento en el servidor**

   Una vez que los datos son recibidos por el servidor Flask, se procesan y almacenan temporalmente en el servidor para permitir la visualización en tiempo real y, posteriormente, el guardado en un archivo CSV. El servidor también se encarga de gestionar los comandos enviados desde el panel web, permitiendo que el usuario decida cuándo guardar los datos.

   El siguiente fragmento de código en el servidor Flask muestra cómo se gestionan los datos y se preparan para ser guardados:

```
function saveFile() {
            const fileName = document.getElementById('fileName').value;
            if (fileName) {
                saveFileContainer.classList.add('hidden');
                fetch(`/download_csv?filename=${fileName}`)
                    .then(response => response.blob())
                    .then(blob => {
                        saveAs(blob, `${fileName}.csv`);
                    })
                    .catch(error => console.error('Error:', error));
            } else {
                alert('Por favor, introduzca un nombre para el archivo.');
            }
        }
```

Código Fuente 5.21: Web panel guardar CSV

En este fragmento:

- `fileName`: Recoge el nombre del archivo proporcionado por el usuario.
- `get_data_from_session()`: Obtiene los datos almacenados temporalmente en la sesión del servidor.
- `csv.writer`: Es el módulo de Python que se utiliza para escribir los datos en el archivo CSV.

5.6.3. Características del archivo CSV

- **Estructura tabular:** Los datos se almacenan en un formato tabular, donde cada fila contiene los valores de un momento determinado, y cada columna representa un tipo de dato (aceleración o velocidad angular).
- **Compatibilidad universal:** El formato CSV es fácilmente accesible en diferentes plataformas, lo que permite que los datos puedan ser utilizados en una variedad de software, desde hojas de cálculo hasta programas de análisis de datos.

5.6.4. Nomenclatura de archivos CSV

Para organizar los archivos generados de manera clara y consistente, se ha diseñado una nomenclatura específica para los archivos CSV al momento de su guardado. Esta nomenclatura sigue el formato:

`Fecha_Usuario_Operacion_Instancia.csv`

- **Fecha:** Indica el día en que se realiza la captura, siguiendo el formato `'YYYYMMDD'`.
- **Usuario:** Indica quién es el usuario que está realizando los movimientos de la raqueta. *Rango: 3 cifras (por ejemplo, 001, 002, 003).*
- **Operación:** Indica qué tipo de operación se está registrando:

00	Calibración
01	Prueba
02	Movimiento Libre
03	Saque
...	...

Tabla 5.1: Tabla de operaciones

Rango: 2 cifras (por ejemplo, 00, 01, 02).

- **Instancia:** Indica qué número de muestra es. *Rango: 2 cifras (por ejemplo, 01, 02, 03).*

Ejemplo: Un archivo guardado con el nombre `2023-10-05_001_02_01.csv` indica que:

- La captura se realizó el 5 de octubre de 2023.
- El usuario identificado con el código `001` realizó la operación.
- Se registró una operación de tipo `02`, que corresponde a ”Movimiento Libre”.
- Este archivo corresponde a la primera instancia de la muestra.

De esta forma, la nomenclatura permite identificar fácilmente la información contenida en cada archivo, facilitando su organización y búsqueda.

Capítulo 6

Resultados y discusión

6.1. Pruebas realizadas

6.1.1. Descripción de las pruebas

Las pruebas de validación se realizaron tomando como referencia las dimensiones estándar de una cancha de bádminton, cuya área es de 6,1 metros de ancho por 13,4 metros de largo, dividiendo la cancha en dos mitades de 6,7 metros para cada jugador. El receptor de datos, que recoge la información transmitida por los sensores MEMS integrados en la raqueta, se encuentra ubicado en la esquina superior izquierda de la cancha, a 2 metros de distancia fuera de la línea de fondo.

Proceso de obtención de las longitudes:

1. **Distancia mínima y máxima dentro del campo:**

   - El cálculo de las distancias se realizó tomando como referencia la ubicación del receptor en la esquina superior izquierda del campo, fuera de la línea de fondo. La distancia mínima, cuando el actor se encuentra en la línea de fondo más cercana al receptor, es de 2 metros.
   - La distancia máxima no es simplemente la longitud del campo, sino la diagonal de la cancha, ya que el actor puede estar en la esquina opuesta, es decir, en la esquina inferior derecha del campo. Para calcular la distancia máxima (diagonal), se aplica el teorema de Pitágoras:

$$\text{Distancia diagonal} = \sqrt{(6,1\,\text{m} + 2\,\text{m} + 0,5\,\text{m})^2 + (6,7\,\text{m} + 0,5\,\text{m})^2} = 11,2\,\text{m}$$

   Esta distancia de 11,2 metros es la distancia máxima dentro del campo, sin incluir el margen adicional fuera de los límites.

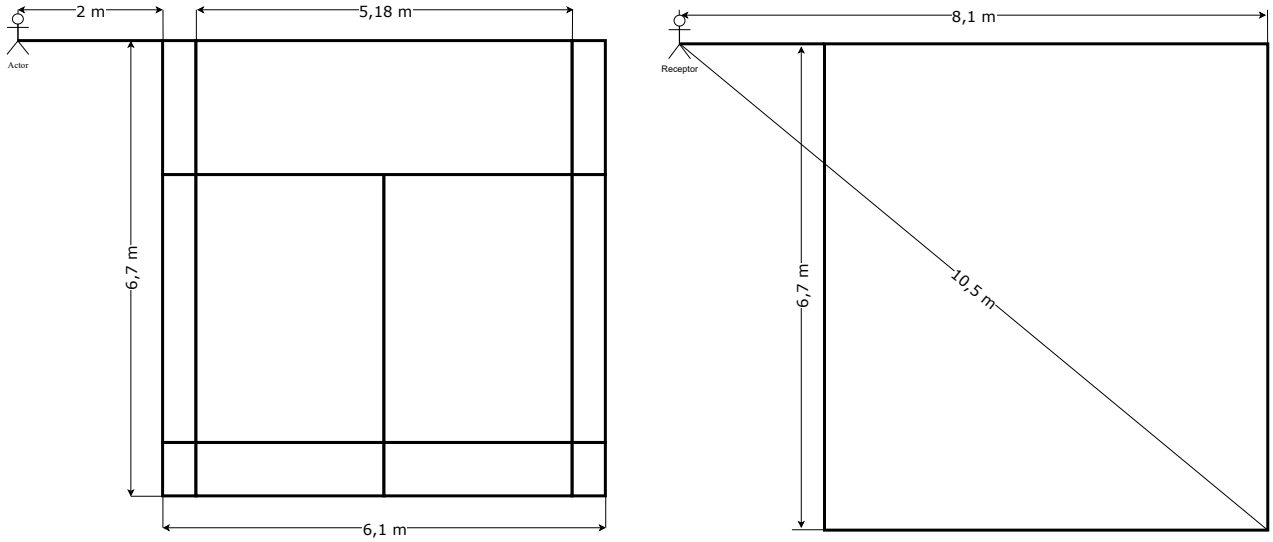

Figura 6.1: Medidas campo badmintón

2. **Distancias de prueba:**

   Para las pruebas de comprobación, se seleccionaron las distancias de 3, 5, 7, 9 y 11 metros con el fin de cubrir un rango progresivo de distancias realistas en las que la raqueta podría estar en relación al receptor en un entorno de juego. Esta decisión está basada en los siguientes puntos:

   a) **Cobertura completa del área de juego:** La cancha de bádminton tiene dimensiones de 6,1 metros de ancho y 6,7 metros de largo. Esto significa que la distancia máxima que puede haber entre la raqueta y el receptor, en una configuración diagonal (esquina inferior derecha a esquina superior izquierda), es aproximadamente 11,2 metros, calculada mediante el teorema de Pitágoras. Las pruebas en incrementos de 2 metros permiten evaluar el rendimiento del sistema tanto en situaciones cercanas como en aquellas que cubren toda la longitud de la cancha.

   b) **Incrementos consistentes:** El aumento en incrementos de 2 metros (3, 5, 7, 9, 11) asegura que haya una progresión uniforme y permite evaluar cómo el sistema se comporta a distancias crecientes. Esto proporciona un análisis comparativo preciso de la calidad de transmisión de datos y la estabilidad del sistema en diferentes puntos.

   c) **Consideración de márgenes:** En las pruebas, también se añaden márgenes de 0,5 metros

adicionales, ya que la raqueta puede sobresalir ligeramente del área reglamentaria durante el juego, especialmente en situaciones límite o golpes extremos. Por ello, la distancia máxima de 11 metros incluye estos márgenes de seguridad.

A continuación, se detallan las principales pruebas:

- **Pruebas a diversas distancias dentro de la cancha:**
  - **Objetivo:** Evaluar la eficiencia del sistema para transmitir los datos recogidos por los sensores MEMS a diferentes distancias, en un rango entre 3 metros y 11,2 metros.
  - **Metodología:** El jugador se posiciona en diferentes puntos de la cancha, comenzando desde los 3 metros y llegando hasta los 11,2 metros, para simular golpes a lo largo del juego. Se analiza la consistencia de la transmisión de datos en función de la distancia entre la raqueta y el receptor de datos ubicado fuera de la cancha. Los datos de aceleración y velocidad angular se capturaron y visualizaron en tiempo real.
  - **Resultados esperados:** Un sistema robusto debe mantener la integridad de los datos, sin pérdida de paquetes o interrupciones en la transmisión a las diferentes distancias, especialmente en el punto más alejado de la cancha.

Esta disposición y metodología permitió comprobar la capacidad del sistema para operar eficazmente en condiciones reales de juego, asegurando que incluso en el caso más desfavorable (el punto más distante), la transmisión de datos seguía siendo fiable.

6.2. Resultados obtenidos

6.2.1. Prueba a 3 metros

Descripción de la prueba: El usuario se sitúa a una distancia de 3 metros del receptor y se registran 4 series del movimiento de la raqueta durante 10 segundos cada una. **Series de la prueba**

- **Serie 3M.1:** Número de muestras: 8560, Archivo: 2024-12-31_001-000-01.csv.
- **Serie 3M.2:** Número de muestras: 8560, Archivo: 2024-12-31_001-000-02.csv.
- **Serie 3M.3:** Número de muestras: 8560, Archivo: 2024-12-31_001-000-03.csv.
- **Serie 4M.4:** Número de muestras: 8560, Archivo: 2024-12-31_001-000-04.csv.

Número de serie	$\min(d_i)(ms)$	$\max(d_i)(ms)$	$\overline{d_i}(ms)$	$\hat{d}_i(ms)$	$\overline{\sigma_i}(ms)$
Serie 3M.1	0	5	1.1667	1	0.9046
Serie 3M.2	0	5	1.1664	1	0.9029
Serie 3M.3	0	5	1.1660	1	0.9045
Serie 3M.4	0	5	1.1659	1	0.8976

Tabla 6.1: Estadísticas de los intervalos entre muestras (Archivos 1-4).

En la Tabla 6.1 se presentan las estadísticas de los intervalos entre muestras consecutivas (d_i) para diferentes series de medición a una distancia de 3 metros. Cada variable en la tabla tiene el siguiente significado:

- $d_i(ms)$: Representa el tiempo transcurrido entre dos muestras consecutivas en milisegundos. Este valor es clave para evaluar la frecuencia de muestreo del sistema.
- $\min(d_i)(ms)$: Indica el intervalo mínimo registrado entre dos muestras consecutivas en cada serie. En este caso, es 0 ms en todas las series, lo que sugiere que en algunos momentos la captura de datos se realizó en intervalos muy cortos o con alguna redundancia en la toma de datos.
- $\max(d_i)(ms)$: Representa el intervalo máximo registrado entre dos muestras consecutivas. En todas las series, este valor es de 5 ms, lo que sugiere que la variabilidad en los tiempos de muestreo es limitada a este rango.
- $\overline{d_i}(ms)$: Es el valor promedio de los intervalos entre muestras dentro de cada serie. Se observa que este valor es aproximadamente 1.166 ms para todas las series, lo que indica que la mayoría de los intervalos entre muestras se encuentran en torno a este valor.
- $\hat{d}_i(ms)$: Representa el intervalo entre muestras más frecuente dentro de cada serie. En todas las series, el valor más recurrente es 1 ms, lo que implica que la mayoría de los datos se registraron con este intervalo de tiempo.
- $\overline{\sigma_i}(ms)$: Es la desviación estándar de los intervalos entre muestras. Este valor mide la variabilidad en los tiempos entre muestras dentro de cada serie. Se observa que la desviación estándar es aproximadamente 0.9 ms en todas las series, lo que sugiere una dispersión moderada en la toma de datos.

En general, los datos presentados en la tabla permiten evaluar la estabilidad del sistema de adquisición de datos en términos de frecuencia de muestreo y variabilidad en los tiempos entre muestras. La distribución de estos intervalos se puede visualizar en la figura 6.2, que muestra la cantidad de ocurrencias de cada intervalo de tiempo registrado en las mediciones.

A partir de este punto, esta nomenclatura se empleará en el resto de tablas para mantener la coherencia en la presentación y análisis de los resultados.

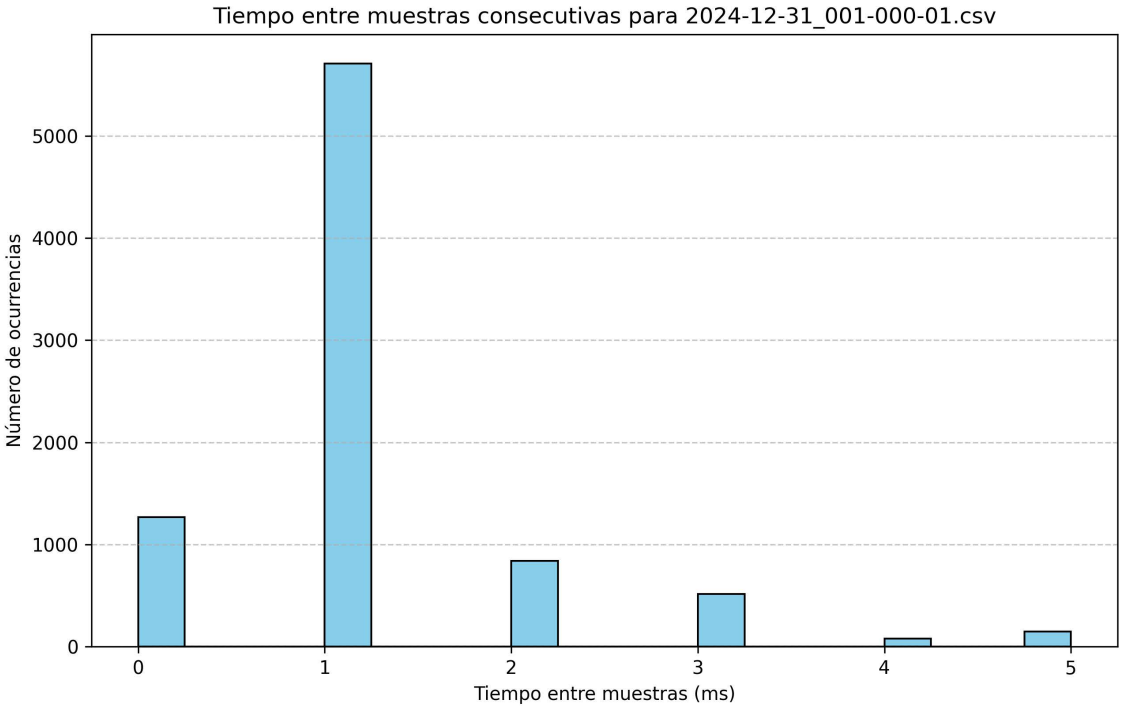

Figura 6.2: Gráfica a 3 metros

6.2.2. Prueba a 5 metros

- **Descripción de la prueba:** El usuario se sitúa a una distancia de 5 metros del receptor y se registran 4 series del movimiento de la raqueta durante 10 segundos cada una.
- **Series de la prueba**
  - **Serie 5M.1**: Número de muestras: 8560, Archivo: 2024-12-31 001-000-05.csv
  - **Serie 5M.2**: Número de muestras: 8560, Archivo: 2024-12-31 001-000-06.csv
  - **Serie 5M.3**: Número de muestras: 8560, Archivo: 2024-12-31 001-000-07.csv
  - **Serie 5M.4**: Número de muestras: 8560, Archivo: 2024-12-31 001-000-08.csv

Número de serie	$\min(d_i)(ms)$	$\max(d_i)(ms)$	$\overline{d_i}(ms)$	$\hat{d}_i(ms)$	$\overline{\sigma_i}(ms)$
5M.1	0	5	1.1668	1	0.9095
5M.2	0	5	1.1664	1	0.8924
5M.3	0	5	1.1665	1	0.9052
5M.4	0	5	1.1664	1	0.9065

Tabla 6.2: Estadísticas de los intervalos entre muestras (Archivos 5-8).

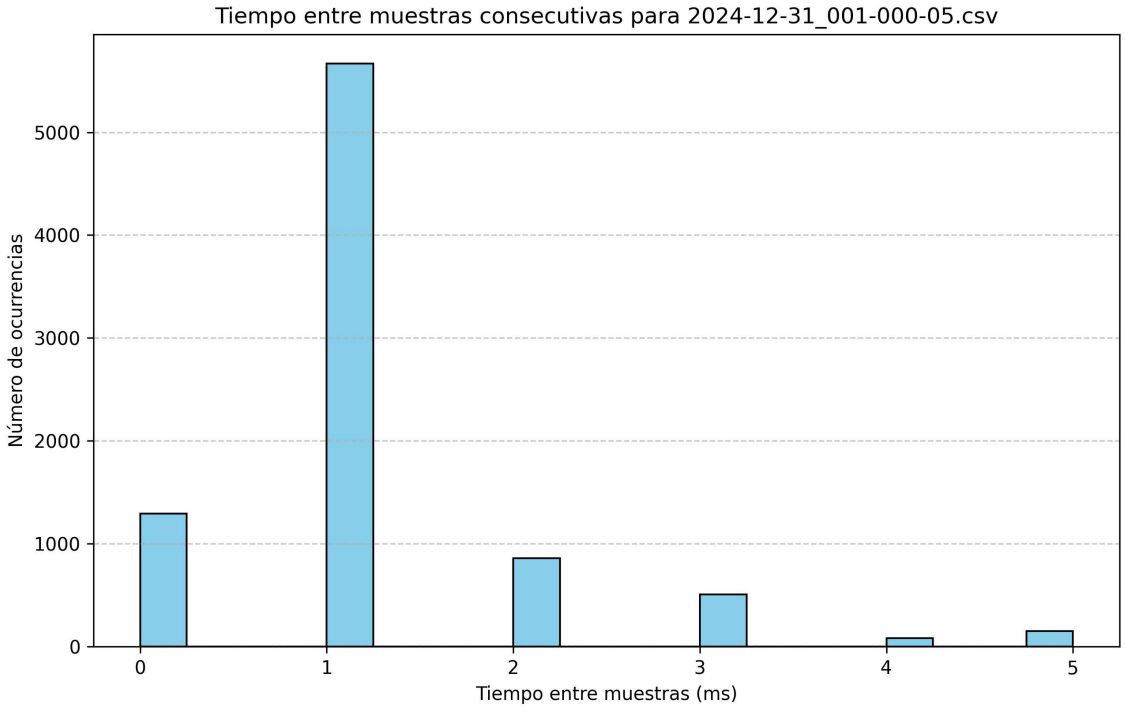

Figura 6.3: Gráfica a 5 metros

6.2.3. Prueba a 7 metros

- **Descripción de la prueba:** El usuario se sitúa a una distancia de 7 metros del receptor y se registran 4 series del movimiento de la raqueta durante 10 segundos cada una.
- **Series de la prueba**
  - **Serie 7M.1:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-09.csv
  - **Serie 7M.2:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-10.csv
  - **Serie 7M.3:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-11.csv
  - **Serie 7M.4:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-12.csv

Número de serie	$\min(d_i)(ms)$	$\max(d_i)(ms)$	$\overline{d_i}(ms)$	$\hat{d}_i(ms)$	$\overline{\sigma_i}(ms)$
Serie 7M.1	0	6	1.1661	1	0.9015
Serie 7M.2	0	5	1.1659	1	0.9070
Serie 7M.3	0	5	1.1660	1	0.9018
Serie 7M.4	0	6	1.1661	1	0.9030

Tabla 6.3: Estadísticas de los intervalos entre muestras (Archivos 9-12).

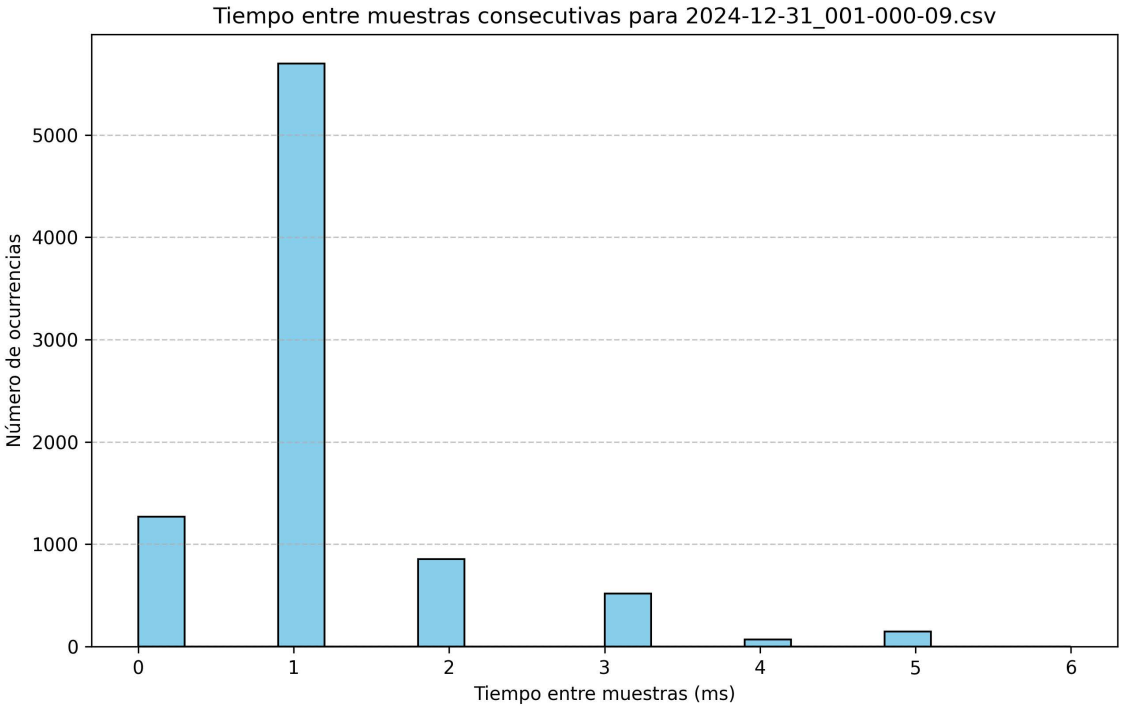

Figura 6.4: Gráfica a 7 metros

6.2.4. Prueba a 9 metros

- **Descripción de la prueba:** El usuario se sitúa a una distancia de 9 metros del receptor y se registran 4 series del movimiento de la raqueta durante 10 segundos cada una.
- **Series por segundo (MPS):**
  - **Serie 9M.1:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-13.csv
  - **Serie 9M.2:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-14.csv
  - **Serie 9M.3:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-15.csv
  - **Serie 9M.4:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-16.csv

Número de serie	$\min(d_i)(ms)$	$\max(d_i)(ms)$	$\overline{d_i}(ms)$	$\hat{d}_i(ms)$	$\overline{\sigma_i}(ms)$
Serie 9M.1	0	5	1.1663	1	0.9005
Serie 9M.2	0	5	1.1661	1	0.9005
Serie 9M.3	0	5	1.1660	1	0.9085
Serie 9M.4	0	5	1.1659	1	0.8995

Tabla 6.4: Estadísticas de los intervalos entre muestras (Archivos 13-16).

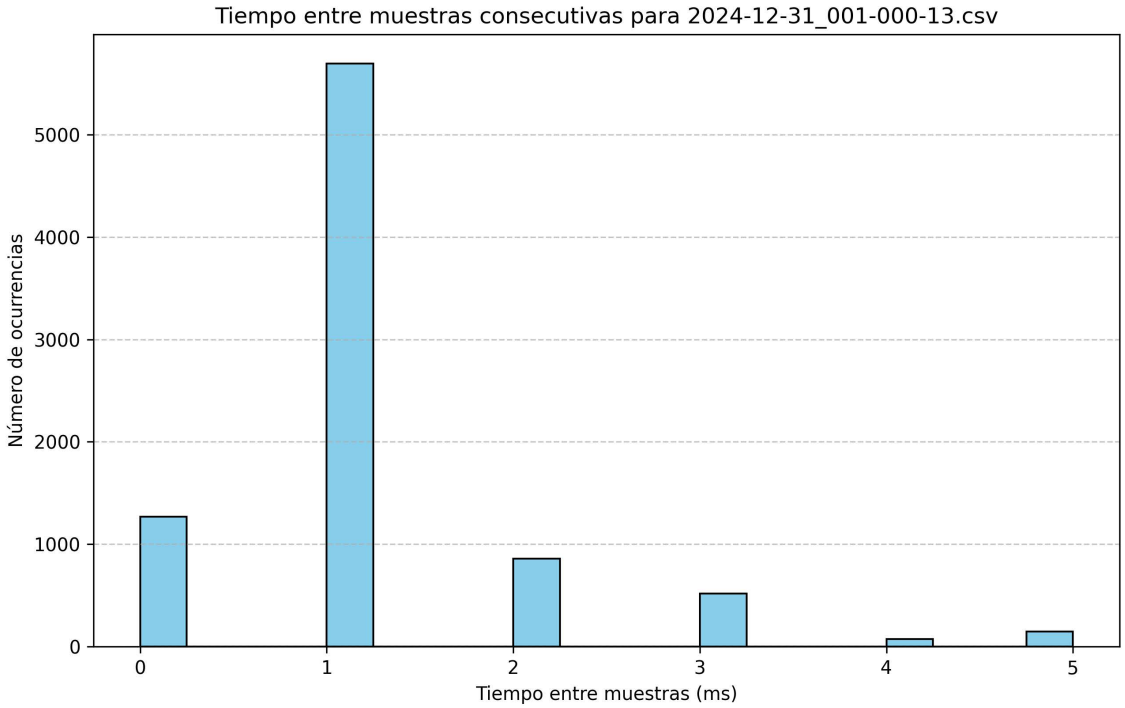

Figura 6.5: Gráfica a 9 metros

6.2.5. Prueba a 11,2 metros

- **Descripción de la prueba:** El usuario se sitúa a una distancia de 11,2 metros del receptor y se registran 4 series del movimiento de la raqueta durante 10 segundos cada una.
- **Series por segundo (MPS):**
  - **Serie 11M.1:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-17.csv
  - **Serie 11M.2:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-18.csv
  - **Serie 11M.3:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-19.csv
  - **Serie 11M.4:** Número de muestras: 8560, Archivo: 2024-12-31 001-000-20.csv

Número de serie	$\min(d_i)(ms)$	$\max(d_i)(ms)$	$\overline{d_i}(ms)$	$\hat{d}_i(ms)$	$\overline{\sigma_i}(ms)$
Serie 11M.1	0	5	1.1660	1	0.9049
Serie 11M.2	0	5	1.1661	1	0.8991
Serie 11M.3	0	5	1.1660	1	0.9002
Serie 11M.4	0	5	1.1660	1	0.9034

Tabla 6.5: Estadísticas de los intervalos entre muestras (Archivos 17-20).

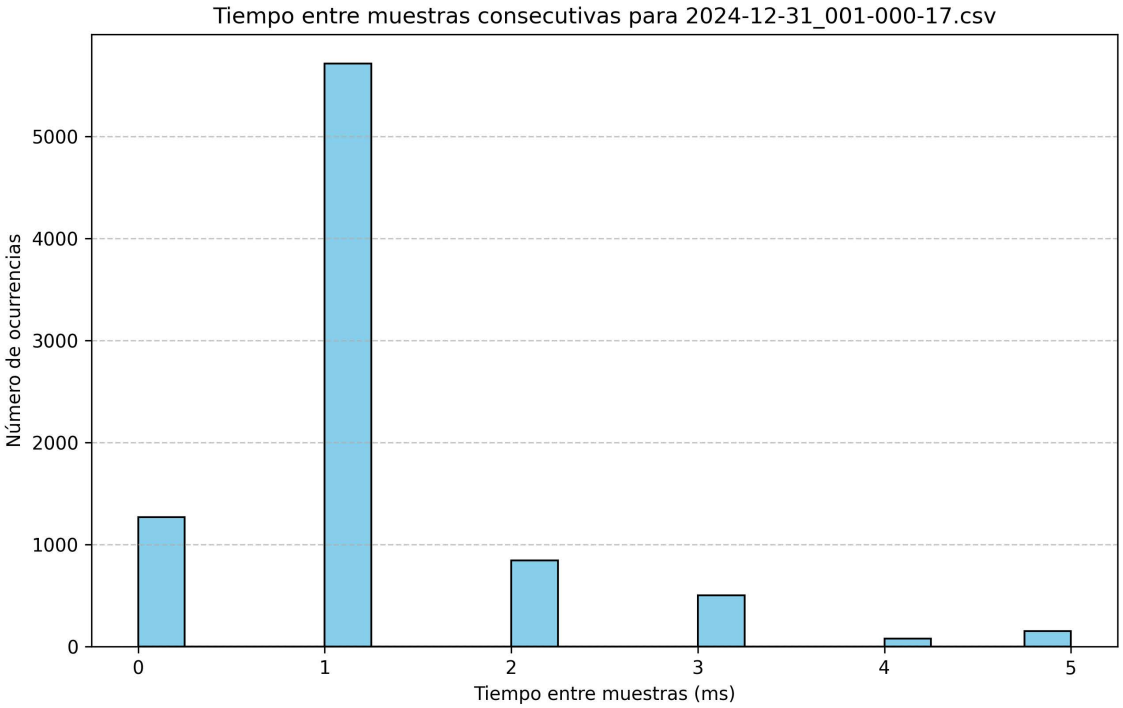

Figura 6.6: Gráfica a 11,2 metros

6.3. Discusión de los resultados

Los resultados obtenidos en las pruebas realizadas destacan la estabilidad y consistencia del sistema desarrollado, logrando mantener una tasa de muestreo constante de 856 muestras por segundo en todas las distancias evaluadas, desde 3 metros hasta 11,2 metros. Este rendimiento homogéneo refleja la robustez tanto del hardware empleado, particularmente la placa Arduino Nano IoT 33, como del software diseñado, con el servidor Flask gestionando la recepción, procesamiento y visualización de datos en tiempo real.

Análisis por Distancias

Aunque las pruebas abarcaron diferentes distancias, el sistema mostró un desempeño uniforme. Esto puede atribuirse al diseño eficiente del circuito, a la conversión de datos de float a bytes para optimizar la transmisión y a la integración de tecnologías inalámbricas estables, como el WiFi basado en el chip NINA-W102.

1. **Cercanía (3 y 5 metros)**: En estas distancias más cortas, no se observaron pérdidas de datos o fluctuaciones en el rendimiento. Esto demuestra la capacidad del sistema para operar en entornos ideales con baja interferencia electromagnética y un alto nivel de conectividad.

2. **Distancias Intermedias (7 y 9 metros)**: En estas condiciones, el sistema mantuvo la misma tasa de muestras, lo que valida la capacidad del diseño para operar eficientemente en un rango típico de uso durante entrenamientos o partidos de bádminton.

3. **Distancia Máxima (11,2 metros)**: Incluso en el límite del alcance experimental, el sistema mantuvo su rendimiento. Esto confirma que las tecnologías utilizadas en la transmisión y recepción de datos son adecuadas para escenarios más exigentes en términos de rango.

Evaluación de los Componentes y su Impacto

- **Placa Arduino Nano IoT 33**: Su capacidad para procesar datos en tiempo real y transmitirlos de manera inalámbrica fue clave para evitar fluctuaciones. La elección de esta placa asegura un balance adecuado entre rendimiento y consumo energético, lo que es esencial en un dispositivo portátil como una raqueta.
- **Servidor Flask**: El procesamiento y almacenamiento de datos gestionados por este servidor demostraron ser eficaces y fiables. El diseño de la arquitectura asegura que no haya cuellos de botella en la recepción y visualización de datos, garantizando una experiencia fluida para el usuario.

Comparación con los Objetivos Iniciales

Los resultados obtenidos cumplen con los objetivos propuestos al inicio del proyecto. Se logró una captura de datos en tiempo real, con visualización inmediata y almacenamiento opcional en archivos CSV, ofreciendo al usuario la posibilidad de analizar los datos según sus necesidades. Además, la estabilidad en el rendimiento asegura que el sistema pueda emplearse no solo en análisis biomecánicos, sino también en escenarios de entrenamiento profesional.

Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

El desarrollo de este sistema de adquisición de datos basado en sensores MEMS integrados en una raqueta de bádminton ha demostrado ser robusto y eficiente en múltiples aspectos, logrando capturar y transmitir datos en tiempo real de forma fiable. Desde su diseño, implementación y pruebas, los resultados obtenidos indican que el sistema cumple satisfactoriamente con los objetivos planteados al inicio del proyecto.

- **Fiabilidad del sistema**: Las pruebas realizadas a diferentes distancias (3, 5, 7, 9 y 11,2 metros) confirman que el sistema es capaz de operar en las condiciones más extremas de un entorno de juego real. La transmisión de los datos a través del servidor Flask demostró ser constante y sin pérdidas significativas de muestras, incluso a distancias superiores a los 11 metros. Este comportamiento resalta la solidez del sistema y su capacidad para soportar variaciones en la posición del jugador en la cancha sin comprometer la integridad de los datos capturados.

- **Integración exitosa del hardware y software**: El uso de la placa Arduino Nano IoT 33, junto con los sensores MEMS, permitió una adquisición precisa de datos de aceleración y velocidad angular. La comunicación con el servidor Flask, a través de WiFi, funcionó sin contratiempos, logrando procesar y visualizar los datos en tiempo real mediante el panel web. Este sistema demostró ser intuitivo para el usuario, lo que facilita su utilización en escenarios deportivos de alta exigencia.

- **Análisis biomecánico en tiempo real**: El sistema desarrollado tiene un gran potencial en el análisis biomecánico de los movimientos durante el juego. Gracias a la visualización en gráficos en tiempo real, los datos recogidos pueden ser analizados para optimizar las técnicas del jugador, mejorar su rendimiento y reducir el riesgo de lesiones. Este tipo de herramienta puede ser utilizada tanto por entrenadores como por jugadores para evaluar sus movimientos y ajustar su estrategia en tiempo real.

- **Versatilidad y potencial para expansión**: A lo largo del proyecto, se ha evidenciado que el sistema puede ser expandido y adaptado a otros deportes o contextos similares. Aunque se ha diseñado específicamente para el bádminton, las funcionalidades del sistema podrían extenderse a otros

entornos que requieran captura de datos en tiempo real con sensores MEMS. Adicionalmente, la infraestructura del sistema permite futuras actualizaciones y mejoras, tales como la integración de algoritmos de aprendizaje automático para el análisis predictivo de los datos capturados.

7.1.1. Seguimiento del Proyecto

A lo largo del desarrollo del proyecto, se ha realizado un seguimiento detallado de las horas invertidas en cada etapa clave del mismo. Este seguimiento ha permitido evaluar tanto la duración total del proyecto como el coste asociado a las diferentes fases. A continuación, se presenta un desglose de las horas estimadas para cada etapa y el tiempo real invertido en cada una:

- Análisis de Requisitos:
  - Horas estimadas: 40 horas
  - Horas reales invertidas: 45 horas
  - Esta etapa incluyó la recopilación de los requisitos funcionales y no funcionales, así como la identificación de los elementos clave del sistema.
- Diseño del Sistema:
  - Horas estimadas: 60 horas
  - Horas reales invertidas: 70 horas
  - El diseño de la arquitectura y la estructura del sistema, junto con la creación de diagramas, tomó más tiempo de lo previsto debido a la necesidad de hacer ajustes a medida que se avanzaba.
- Desarrollo de Prototipos:
  - Horas estimadas: 160 horas
  - Horas reales invertidas: 190 horas programación y configuración de los prototipos implicaron una mayor cantidad de horas, principalmente debido a las dificultades encontradas en la integración del hardware y el software, lo que retrasó el progreso.
- Pruebas y Validación:
  - Horas estimadas: 80 horas
  - Horas reales invertidas: 100 horas
  - Las pruebas y validación fueron un proceso iterativo, que implicó ajustes en el sistema para garantizar su fiabilidad y precisión. Esto resultó en un tiempo superior al estimado.
- Documentación y Entrega:
  - Horas estimadas: 60 horas

- Horas reales invertidas: 70 horas
- La documentación fue más extensa de lo planeado debido a la necesidad de detallar todos los aspectos técnicos y de usuario, lo que contribuyó al tiempo adicional.

Total de horas estimadas: 400 horas
Total de horas reales invertidas: 475 horas

Duración del proyecto:

La duración total del proyecto se extendió durante 7 meses, con un retraso de aproximadamente 1 mes respecto al cronograma inicial. Los retrasos se deben principalmente a la parte de desarrollo del software y las pruebas de testeo, que requirieron más tiempo debido a los desafíos encontrados en la integración de componentes y la validación del sistema en condiciones reales.

Coste total:

El coste real del proyecto se puede desglosar considerando tanto el tiempo de desarrollo como los materiales utilizados. Asumiendo una tarifa estándar de 20€/hora para un perfil técnico en este tipo de proyectos, el coste asociado a la dedicación temporal sería:

$$475 \text{ horas} \times 20 \text{ €/hora} = 9.500 \text{ €}$$

Si en su lugar se aplicara una tarifa de 30€/hora, acorde con el valor de mercado de un ingeniero junior, el coste estimado ascendería a:

$$475 \text{ horas} \times 30 \text{ €/hora} = 14.250 \text{ €}$$

A esto se suman los gastos en materiales y hardware utilizado para la implementación:

- Placa Arduino Nano 33 IoT: 30€
- Batería y regulador de voltaje: 20€
- Material para la impresión 3D del mango: 15€
- Cables y componentes electrónicos adicionales: 10€

En total, el coste del hardware asciende aproximadamente a 75€, por lo tanto, el coste total estimado del proyecto se sitúa entre 9.575€ y 14.325€, dependiendo de la valoración de la mano de obra empleada en su desarrollo.

En resumen, aunque hubo un desajuste en los tiempos estimados debido a retrasos en el desarrollo del software y las pruebas de testeo, el seguimiento detallado de las horas invertidas ha permitido obtener una visión clara del esfuerzo y los recursos necesarios para el desarrollo exitoso del proyecto. El sistema ha cumplido con los objetivos planteados, proporcionando una plataforma sólida para la captura, procesamiento y análisis de datos en tiempo real. Las pruebas realizadas validan la capacidad del sistema para operar eficazmente en un entorno de juego real, y las oportunidades de mejora y expansión abren las puertas a aplicaciones más amplias y ambiciosas.

7.2. Líneas futuras

El presente proyecto ha logrado desarrollar un sistema de captura de datos en tiempo real, proporcionando una base sólida para el análisis biomecánico de movimientos durante el juego de bádminton mediante sensores MEMS integrados en una raqueta. No obstante, se identifican una serie de oportunidades para mejorar y expandir las capacidades del sistema. Estas mejoras buscan no solo aumentar la eficiencia y la precisión en la captura y almacenamiento de datos, sino también ofrecer un sistema más flexible y robusto que se adapte a diferentes entornos y necesidades.

En las siguientes líneas futuras se proponen cambios y adiciones que permitirán al sistema evolucionar hacia una infraestructura más escalable, eficiente y orientada al usuario, mejorando tanto su rendimiento técnico como su aplicabilidad en entornos más exigentes.

1. **Migración del servidor a infraestructuras más avanzadas:** Actualmente, el servidor Flask corre en un entorno limitado, gestionando la recepción y procesamiento de datos desde la raqueta de bádminton. Una línea de mejora sería la migración de este sistema a una plataforma más robusta, como una Raspberry Pi, la cual no solo permitiría una mayor portabilidad, sino también mayor capacidad de procesamiento y almacenamiento en un entorno dedicado y económico. Además, se podría optar por servicios en la nube para alojar el servidor, lo que ofrecería escalabilidad y acceso remoto, mejorando la flexibilidad del sistema.

2. **Implementación de una base de datos relacional:** Aunque los datos actualmente se almacenan en archivos CSV, un enfoque más robusto sería la integración de una base de datos como MySQL o PostgreSQL para el almacenamiento de las muestras. Esto permitiría no solo el almacenamiento más eficiente de grandes volúmenes de datos, sino también la posibilidad de realizar consultas avanzadas y obtener informes personalizados según las necesidades del usuario. Además, se podría incluir la funcionalidad de interacción en tiempo real con la base de datos desde el panel web, permitiendo analizar y visualizar datos históricos de manera dinámica.

3. **Optimización y expansión del sistema de visualización de datos:** Si bien el sistema actual permite la visualización de datos en tiempo real mediante gráficos interactivos, una futura mejora podría ser la integración de bibliotecas de visualización más avanzadas, como D3.js, para proporcionar gráficos más detallados y personalizables. Esto facilitaría el análisis de patrones complejos en los datos de aceleración y velocidad angular de la raqueta.

4. **Incorporación de algoritmos de machine learning:** Una línea futura ambiciosa sería la implementación de algoritmos de machine learning para detectar patrones o anomalías en los datos recogidos. Con un modelo entrenado con suficientes datos, el sistema podría sugerir mejoras en la técnica del jugador o detectar comportamientos inusuales durante el juego que puedan requerir atención.

5. **Ampliación de la funcionalidad del panel web:** Otra línea de mejora sería la ampliación del panel web, permitiendo la gestión de usuarios y configuraciones avanzadas para distintos tipos de análisis. Además, se podría agregar la opción de compartir y visualizar los datos en una aplicación móvil, lo que ampliaría considerablemente la usabilidad del sistema en diferentes dispositivos.

Bibliografía

[1] Arduino. Arduino_lsm6ds3 - Arduino Reference, July 2019.

[2] Arduino. WiFi - Arduino Reference, June 2015.

[3] Pallets. Welcome to Flask — Flask Documentation (3.0.x), July 2019.

[4] GitHub. GitHub.com Documentación de la Ayuda, July 2020.

[5] Arduino. WiFiNINA - Arduino Reference, July 2018.

[6] Mermaid.js. About Mermaid | Mermaid, December 2022.

[7] Chart.js. Chart.js | Chart.js, August 2024.

[8] FileSaver.js. FileSaver.js - Libraries - cdnjs, September 2017.

[9] Douglas E. Comer. *Internetworking with TCP IP. 1: Principles, protocols, and architecture*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 2. ed. - 1991 edition, 1991.

[10] Herbert Schildt. *Programación en lenguaje C*, volume 1. Osborne-McGraw-Hill, Madrid, 1st edition, 1990. OCLC: 640026308.

[11] Tim Parker. *Teach yourself TCP/IP in 14 days*, volume 1. Sams Publishing, Indianapolis, IN, 1994.

[12] Herbert Schildt. *C: Manual de referencia*, volume 1. McGraw-Hill, Madrid, España, 1988. OCLC: 61681658.

[13] Niklaus Wirth. *Algoritmos, estructuras de datos, programas*, volume 1. Ediciones del Castillo, Madrid, 5th edition, 1986. OCLC: 758170715.

[14] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX companion*, volume 1 of *Addison-Wesley series on tools and techniques for computer typesetting*. Addison-Wesley, Boston Munich, nachdr. edition, 2003.

[15] Michael Phomsoupha and Guillaume Laffaye. The science of badminton: game characteristics, anthropometry, physiology, visual fitness and biomechanics. *Sports medicine*, 45:473–495, 2015.

[16] Dariusz Tomaszewski, Jacek Rapiński, and Renata Pelc-Mieczkowska. Concept of ahrs algorithm designed for platform independent imu attitude alignment. *Reports on Geodesy and Geoinformatics*, 104(1):33–47, 2017.

Anexo

A.1. Repositorio para el código

El código de este proyecto se encuentra en `https://github.com/Nicolas318/Adquisicion-de-mediciones-fisicas-en-tiempo-real`.

El repositorio contiene los archivos esenciales para la implementación del sistema, incluyendo los scripts del servidor Flask, el código para la captura de datos en la placa Arduino Nano IoT 33 y la interfaz web para la visualización y control del sistema. A continuación, se describe la estructura de carpetas y archivos más importantes:

- **/arduino**

  Esta carpeta contiene el código que debe cargarse en la placa Arduino Nano IoT 33. El código Arduino es responsable de interactuar con los sensores MEMS, capturando datos sobre aceleración, velocidad angular y otras variables físicas de la raqueta. Estos datos se procesan localmente y se envían al servidor UDP a través de una conexión Wi-Fi, utilizando el protocolo de comunicación UDP.

  - **BinarySendUDP.ino**

    Este archivo contiene el código principal que se ejecuta en el Arduino Nano IoT 33. Algunas de las principales funcionalidades de este código incluyen:

    - **Inicialización de los sensores MEMS:** Configura el acelerómetro y el giroscopio integrados para empezar a tomar lecturas en tiempo real.
    - **Captura de datos de los sensores:** Lee los valores del acelerómetro y giroscopio, que corresponden a la aceleración en los ejes X, Y y Z, y las velocidades angulares en los mismos ejes.
    - **Procesamiento de datos:** Convierte los valores capturados a un formato adecuado (como floats) y los empaqueta en formato binario (bytes) para una transmisión eficiente.
    - **Envío de datos mediante UDP:** Los datos procesados se transmiten al servidor Flask utilizando la biblioteca WiFiUDP, que permite enviar paquetes de datos a través de una red Wi-Fi al servidor.

- **/Flask-server**

  Esta carpeta contiene los archivos necesarios para implementar el servidor Flask. Este servidor se encarga de la recepción de los datos enviados por el Arduino, su procesamiento y la visualización a través de una interfaz web interactiva que se comunica con el usuario. Aquí se encuentran todos los componentes del servidor y la interfaz de usuario.

  - **/static**

    Esta carpeta incluye los archivos estáticos utilizados por la interfaz web, como los archivos de estilos CSS que definen el diseño y apariencia de la página web.

    - **styles.css:** Este archivo contiene las reglas CSS que controlan el diseño visual del panel web. Define el estilo de los botones, los gráficos y otros elementos de la interfaz. Permite que la página tenga una apariencia organizada y que sea fácil de usar para el usuario final, proporcionando un diseño limpio y funcional.
    - **Favicon.ico:** El favicon es un pequeño icono que aparece en la pestaña del navegador cuando el usuario accede a la página web del sistema. Se almacena aquí para ser cargado junto con el resto de los elementos estáticos.

  - **/templates**

    Esta carpeta contiene el código de la página web escrita en HTML que el usuario final usará para interactuar con el sistema.

    - **index.html:** Este archivo es el núcleo de la interfaz web, permitiendo al usuario iniciar, detener o realizar pruebas en el sistema de adquisición de datos. A través de esta página, el usuario puede enviar comandos al servidor Flask, los cuales son enviados posteriormente al Arduino para iniciar o detener la captura de datos. También permite la visualización en tiempo real de los datos que son capturados por el Arduino. Además, utiliza gráficos interactivos proporcionados por la biblioteca Chart.js para mostrar los valores de aceleración y giroscopio de forma clara y visualmente atractiva.

  - **pythonReceiver.py**

    Este es el script principal del servidor, escrito en Python utilizando el framework Flask. Este archivo cumple varias funciones críticas en el sistema:

    - **Servidor UDP:** El servidor Flask incluye un servidor UDP que recibe los datos transmitidos por el Arduino. A través de sockets UDP, el servidor escucha en un puerto específico (por ejemplo, el puerto 5005) y recibe los paquetes de datos en formato binario (bytes).
    - **Desempaquetado y porcesamiento de datos:** El servidor toma los datos binarios enviados por el Arduino y los convierte a su formato original (floats). Cada paquete de datos contiene múltiples lecturas del acelerómetro y giroscopio, que son procesadas por el servidor para su visualización en la interfaz web.
    - **Comandos desde el panel web:** El servidor Flask también gestiona las interacciones desde el panel web. Cuando el usuario hace clic en un botón en la página web (como

Ïniciar Captura.º ”Detener Captura”), Flask recibe la solicitud, traduce el comando y lo envía al Arduino para ejecutar la acción correspondiente.

- **Visualización en tiempo real:** El servidor Flask interactúa con el panel web para proporcionar actualizaciones en tiempo real de los datos. Utilizando AJAX (a través de JavaScript), los datos de los sensores son solicitados de manera continua y se actualizan en los gráficos que el usuario puede ver.
- **Almacenamiento de datos:** Después de que los datos han sido capturados y procesados, el servidor tiene la capacidad de guardarlos en archivos CSV para su posterior análisis. Estos archivos almacenan todas las lecturas de los sensores en un formato estructurado que puede ser utilizado para revisiones y estudios detallados del rendimiento.

A.2. Manual de instalación

En este Anexo se describen los pasos necesarios para poner en marcha el sistema implementado en este proyecto. Esta instalacion se ha probado en una máquina con el sistema operativo Windows 11 Home 23H2.

A.2.1. Instalación de Arduino

1. **Conseguir una placa Arduino Nano Iot 33 y el cable USB.**
   - **Comprar una placa:** La forma mas sencilla es comprando una ya lista, puedes conseguir varios modelos y precios aquí.
2. **Descargar el ambiente de desarrollo Arduino**
   - `https://www.arduino.cc/en/software`
3. **Instala el entorno de desarrollo de Arduino (IDE)**
   a) Ejecuta el instalador del programa y sigue los pasos de instalación.
   b) Acepta los términos y condiciones de la licencia.
   c) Selecciona todas las opciones para que instale todos los complementos y drivers necesarios.
   d) Selecciona la ruta de instalación y presiona “install”.
4. **Conectar la tarjeta Arduino**
   - La alimentación puede ser provista por el puerto USB o por una fuente externa (6-12V). Cualquiera sea el modo de alimentación conecta el cable USB al PC.
   - El LED de encendido debería iluminarse.
5. **Ejecutar el ambiente de desarrollo Arduino**

6. **Cargar el programa en la tarjeta**

   a) Descarga el programa de Arduino del repositorio.

   ```
   git clone https://github.com/Nicolas318/
   Adquisicion-de-mediciones-fisicas-en-tiempo-real.git
   ```

   b) Abre el sketch de ejemplo: Archivo >Abrir... >SendBinaryUDP.ino

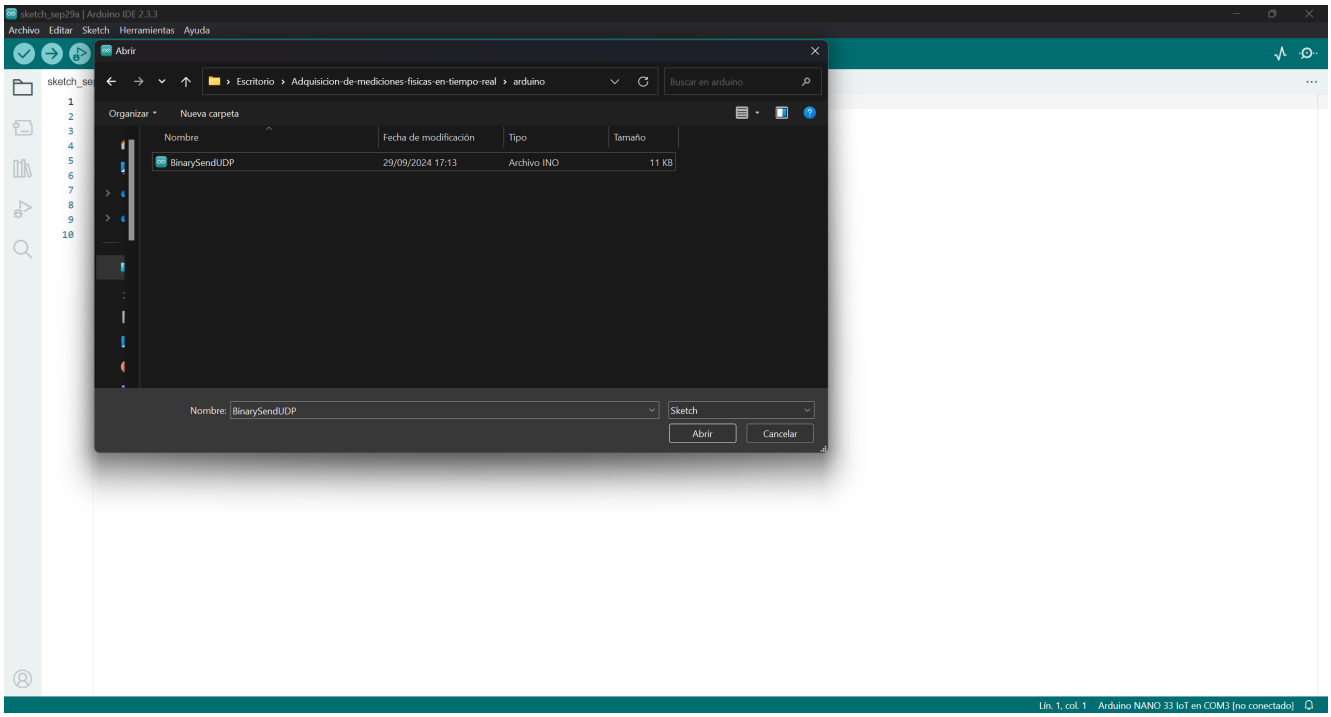

Figura A.1: Abrir Sketch

   c) Selecciona el puerto COM en el que tienes conectada la tarjeta Arduino en el menú Herramientas >Puerto.

   d) Necesitarás también especificar el microcontrolador que estás utilizando. Mira el chip que está instalado en tu tarjeta y seleccionalo en Herramientas >Placa:

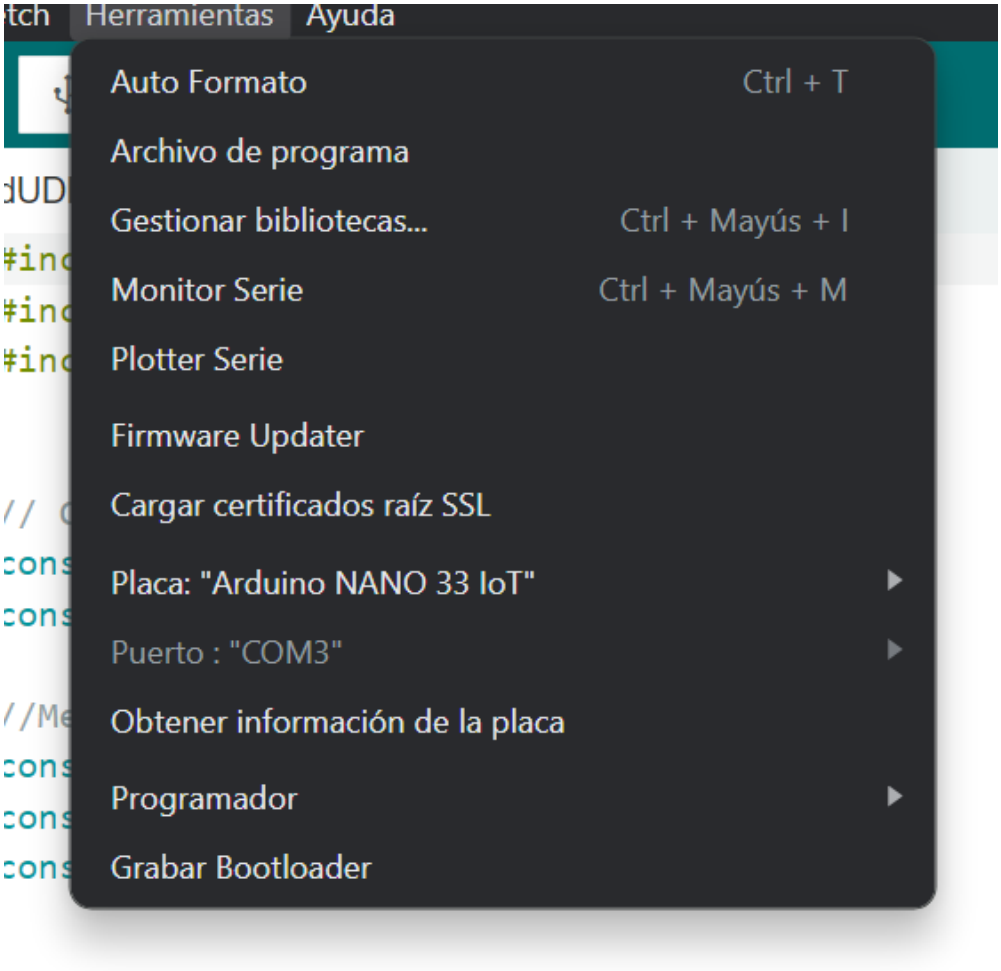

Figura A.2: Puerto y placa

e) Haz click en el botón “Upload”. Si el programa se cargó satisfactoriamente un mensaje aparecerá en la barra de status “Done uploading”.

A.2.2. Instalación de Python

1. **Descargar Python**
   - Ve a la página oficial de Python: `https://www.python.org/downloads/` y descarga la versión más reciente de Python.
2. **Instalar Python**
   - Ejecuta el instalador y sigue los pasos:
     a) Asegúrate de marcar la opción `Add Python to PATH` durante la instalación para poder ejecutar Python desde la terminal o el símbolo del sistema.
     b) Sigue los pasos de instalación predeterminados y finaliza la instalación.
3. **Verificar la instalación**

- Abre una terminal o el símbolo del sistema y escribe `python --version`. Si la instalación fue exitosa, deberías ver la versión de Python que instalaste.

4. **Instalar Flask**

   - Flask es un framework de Python necesario para ejecutar el servidor. Para instalarlo, ejecuta el siguiente comando en la terminal:

     ```
     pip install flask
     ```

5. **Instalar otras dependencias**

   - Dependiendo del proyecto, es posible que necesites instalar otras bibliotecas necesarias para el correcto funcionamiento del sistema. En este caso, el archivo requirements.txt dentro del repositorio del proyecto especifica las dependencias adicionales. Ejecuta el siguiente comando para instalarlas:

     ```
     pip install -r requirements.txt
     ```

A.3. Puesta en marcha del sistema

Una vez instalados los componentes de software y hardware descritos anteriormente, la puesta en marcha del sistema sigue una serie de pasos que garantizan la comunicación entre la raqueta, el servidor y la interfaz web. A continuación se detallan los pasos para poner en marcha el sistema:

1. **Encendido del hardware:**

   a) **Activar la placa Arduino Nano IoT 33:** Enciende la placa utilizando el interruptor conectado a la batería que alimenta la placa.

   b) **Verificación de la conexión a la red WiFi:** La placa debe conectarse automáticamente a la red WiFi configurada. Puedes verificarlo observando el comportamiento del LED de estado o a través de mensajes de depuración en el monitor serie si tienes acceso a un puerto USB.

2. **Inicio del servidor Flask y script de recepción**

   a) Abre una terminal en el directorio donde se encuentra el script `pythonReceiver.py`.

   b) Ejecuta el siguiente comando para iniciar el servidor que recibirá los datos del Arduino:

   ```
   python pythonReceiver.py
   ```

   c) El servidor debería estar listo para recibir los datos vía UDP desde el Arduino y comenzar a procesarlos. Verifica que aparezca un mensaje en la terminal indicando que el servidor está esperando los datos.

3. **Prueba de conectividad con el Arduino**

   a) Una vez que el servidor está en funcionamiento, asegúrate de que el Arduino puede comunicarse correctamente con el servidor Flask a través de UDP. La placa debe enviar un mensaje de "handshake" al servidor. En la terminal donde ejecutaste el script `pythonReceiver.py`, deberías ver un mensaje confirmando la conexión exitosa.

4. **Acceso a la interfaz web**

   a) Abre un navegador web e ingresa la dirección `http://localhost:5000` para acceder al panel de control del sistema.

   b) Desde esta interfaz, el usuario puede interactuar con el sistema, iniciar la captura de datos, realizar pruebas de sensores, o verificar el estado de los datos capturados en tiempo real.

5. **Prueba de captura de datos**

   a) Desde el panel web, selecciona la opción "Captura continua" o "Captura temporizada" y verifica que los datos de aceleración y velocidad angular de los sensores MEMS son recibidos y procesados por el servidor.

   b) Los datos deben visualizarse en gráficos interactivos dentro del panel web. Si los datos son correctos, se pueden almacenar en un archivo CSV para su posterior análisis.

6. **Verificación y ajuste**
   Si se encuentran problemas durante la puesta en marcha, es recomendable revisar:

   - Que la placa Arduino está correctamente conectada a la red WiFi.
   - Que las dependencias necesarias están instaladas (por ejemplo, Flask y otras librerías especificadas en `requirements.txt`).
   - Que el servidor Flask está recibiendo los datos del Arduino sin interrupciones.