

# **Taller de Proyecto II**

## **Informe Final**

### **A1 – Auto VR con ESP32 CAM**

#### **Grupo de Desarrollo**

- Garay Francisco - 02714/4
- Lambre Jerónimo – 02628/7
- Bruno Laureano – 02585/3

# Contenido

Contenido .....	1
Introducción .....	1
1. Descripción General del Proyecto .....	1
1.1. Objetivos primarios .....	1
1.2. Objetivos secundarios .....	2
2. Presentación Esquemática del Proyecto .....	3
2.1. Conexión de Hardware .....	4
2.1.1. Placa Auto .....	4
2.1.2. Placa Torreta .....	6
3. Documentación del Software del Proyecto .....	9
3.1. Bloque de gafas VR y giroscopio .....	9
3.1.1. Estudio del módulo MPU6050 .....	9
3.1.2. Configuración del entorno para trabajar con la placa de desarrollo .....	10
3.1.3. Programación de la placa y el giroscopio .....	11
3.1.4. Modificación del cálculo de rotación .....	12
3.1.5. Calibración del MPU6050 .....	13
3.1.6. Medición de rotación con respecto al eje Z .....	14
3.1.7. Control remoto de la torreta con MPU6050 .....	15
3.2. Servidor central .....	16
3.3. Servidor Web .....	17
3.3.1. Flask Web Server con dos ESP32-CAM diseñado para las gafas VR .....	17
3.3.2. Análisis de las estadísticas de la transmisión de video utilizando TCP .....	19
3.4. Servidor MQTT .....	21
3.4.1. Publicador MQTT a ESP32-CAM que controla el auto .....	22
3.4.2. Suscriptor MQTT desde la ESP32-CAM que controla el auto .....	24
3.5. Control del vehículo .....	26
3.6. Medición de tiempos del protocolo MQTT para ESP32-CAM que controla el movimiento del auto .....	28
4. Documentación Relacionada .....	31
4.1.1. Placas desarrolladas .....	33
Apéndice A: Materiales y Presupuesto .....	37
Apéndice B: Otros protocolos y desarrollos futuros .....	37
B.1 Protocolo RTSP sobre UDP .....	37
B.2 Servidor NTP para la sincronización de los dispositivos de la red .....	40

Apéndice C: Pruebas y desarrollos.....	42
C.1    Desarrollo y configuración de la Torreta .....	42
C.2    Pruebas preliminares con un servomotor .....	44
C.3    Prueba inicial: Desarrollo del WebServer y Streaming de Video con TCP .....	44
C.4    Prueba inicial: Uso del ESP32 Como Access Point .....	48
C.5    Uso del ESP32 conectándose a una red Externa .....	49
C.6    Prueba de funcionamiento de los ESP32-CAM .....	49
C.7    Prototipo: Conexionado físico del vehículo para lograr únicamente su movimiento utilizando MQTT .....	50
C.8    Configuración del Flask Web Server .....	52
C.9    Código del publicador MQTT al auto .....	53
C.10    Código del suscriptor desde la ESP32-CAM que controla el auto.....	53
C.11    Código de la función recall para el control del auto.....	54

## **Introducción**

En el entorno actual de rápida evolución del Internet de las Cosas (IoT) en la industria y el amplio desarrollo que ha tenido la realidad virtual (VR), se plantea una creciente demanda de soluciones que integren ambas tecnologías de forma efectiva y eficiente. A medida que las interfaces hombre-máquina avanzan, el desafío no es solo capturar información del entorno, sino también permitir una interacción intuitiva y en tiempo real con el mismo. En este contexto, se identifica una brecha significativa: la capacidad de controlar sistemas robóticos de manera inmersiva, utilizando una plataforma VR, de tal forma que la interacción sea tan natural como el movimiento humano en sí.

Aunque las interfaces actuales permiten un control efectivo de vehículos y sistemas robóticos, la experiencia podría mejorar con la inclusión de visión estereoscópica del entorno y una mayor inmersión para el operador. Esta inmersión visual puede ser fundamental para tareas que demandan un alto grado de precisión y conciencia espacial, además de ofrecer una experiencia de usuario más rica y atractiva.

El proyecto "Auto con Anteojos de VR con ESP32" busca satisfacer dicha necesidad al ofrecer un sistema integrado que combina un vehículo controlado a distancia, equipado con un brazo robótico y cámaras de visión estereoscópica, con una interfaz de visión mediante gafas VR. Este sistema no solo ofrece una perspectiva en primera persona del entorno a través de un par de cámaras, sino que también permite el control intuitivo del brazo robótico y del vehículo mediante el seguimiento de los movimientos de la cabeza del usuario.

Con este enfoque, el proyecto aspira a transformar la manera en que se interactúa con los sistemas robóticos y vehículos a control remoto, ofreciendo una experiencia más inmersiva y natural. Esta solución tiene aplicaciones potenciales en una variedad de campos, desde la robótica educativa hasta los sistemas de inspección industrial y la exploración de áreas de difícil acceso.

### **1. Descripción General del Proyecto**

El alcance del proyecto es el de desarrollar un sistema que permita al usuario explorar el entorno haciendo uso de un auto robot controlable de forma inalámbrica, y unas gafas de realidad virtual (VR) con las cuales podrá ver desde la perspectiva del auto robot.

#### **1.1. Objetivos primarios**

- Visualizar en las gafas VR la vista desde la perspectiva del auto robot.
- Desarrollar un vehículo controlado a distancia.

- Crear un sistema de control para el vehículo.
- Desplegar una WLAN que conecte de forma inalámbrica los dispositivos del sistema.
- Diseñar e implementar un brazo robótico con dos grados de libertad
- Orientar la posición de la cámara según el movimiento de las gafas VR en dos ejes de libertad utilizando servomotores y un giroscopio.
- Optimizar la latencia en la transmisión de datos y video para mejorar la inmersividad de la experiencia de usuario
- Implementar estrategias para optimizar el consumo de energía del sistema.

## **1.2. Objetivos secundarios**

- Implementar medidas de seguridad para prevenir el acceso no autorizado al sistema de control y a la visualización de las imágenes.

Al priorizar estos objetivos, el proyecto busca entregar un prototipo funcional en los plazos acordados, manteniendo un balance entre complejidad, viabilidad y eficacia del sistema.

## 2. Presentación Esquemática del Proyecto

A continuación, se puede ver un esquema detallado del proyecto en formato gráfico con todas las partes físicas involucradas (Figura 1). En el mismo se identifica cada parte funcional y su relación con las partes físicas

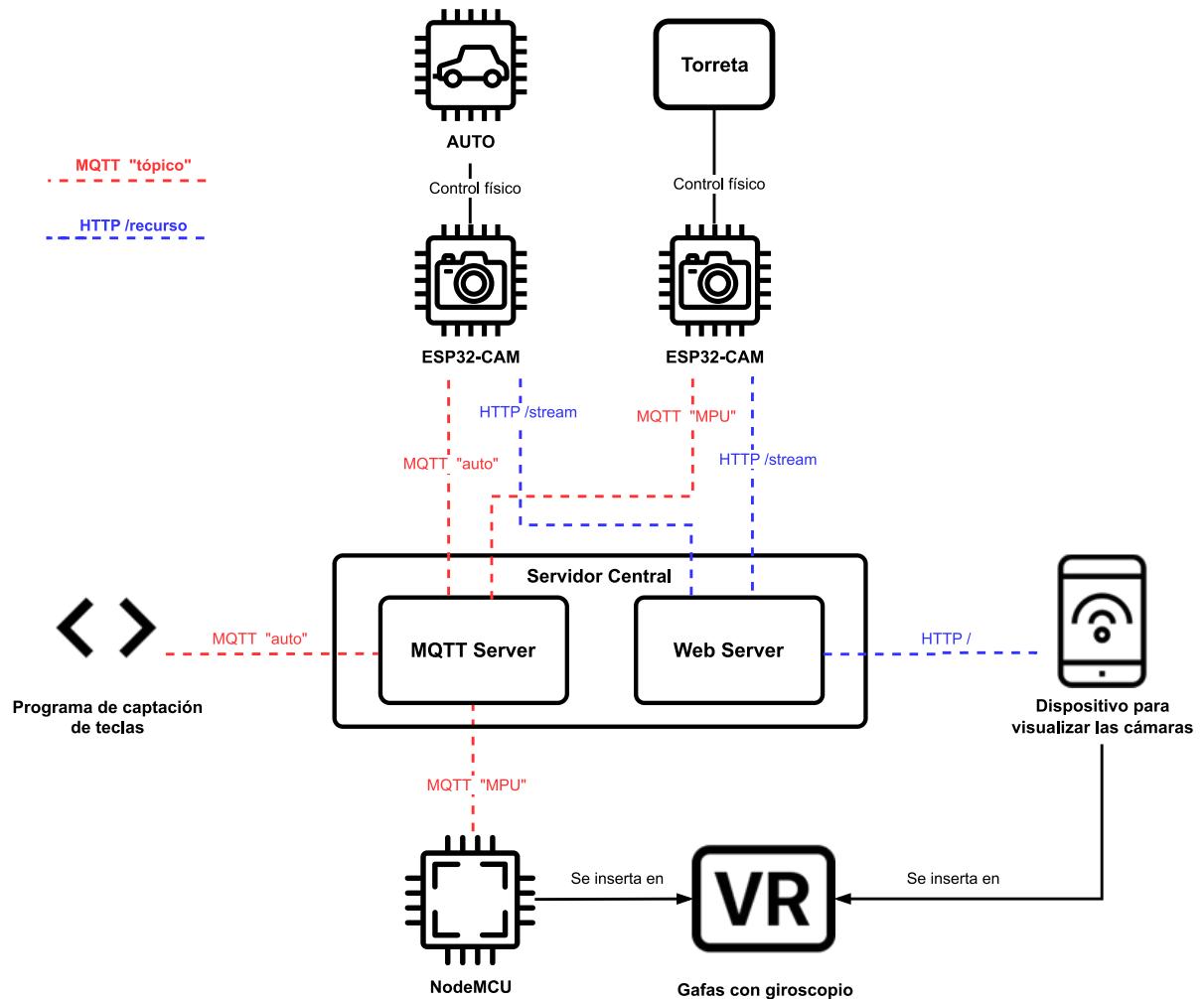


Figura 1 - Esquema gráfico del proyecto

Se planteó una topología estrella para interconectar los múltiples dispositivos de la red. En el centro de la misma se encuentra el servidor central que provee funcionalidades de conexión a todos los módulos del sistema ya que contiene los servidores Web y MQTT. Además, en la Figura 1 se muestran las conexiones según su naturaleza: línea punteada inalámbrica y línea continua alámbrica y/o agregado de pieza. Las conexiones MQTT se muestran con rojo junto al nombre del tópico. Por otro lado, las conexiones HTTP se muestran en azul junto al nombre del recurso que se está accediendo.

## 2.1. Conexión de Hardware

Para una mejor organización de cableado y de espacio en el vehículo, se diseñaron dos PCBs. A continuación, se explicarán con mayor detalle cada una de ellas.

### 2.1.1. Placa Auto

Esta placa fue diseñada para la conexión y control de cuatro servomotores. Recibe señales de modulación de ancho de pulso (PWM) y tensión de entrada. Además, proporciona la alimentación a la “Placa Torreta”.

El diagrama esquemático es el siguiente (Figura 2):

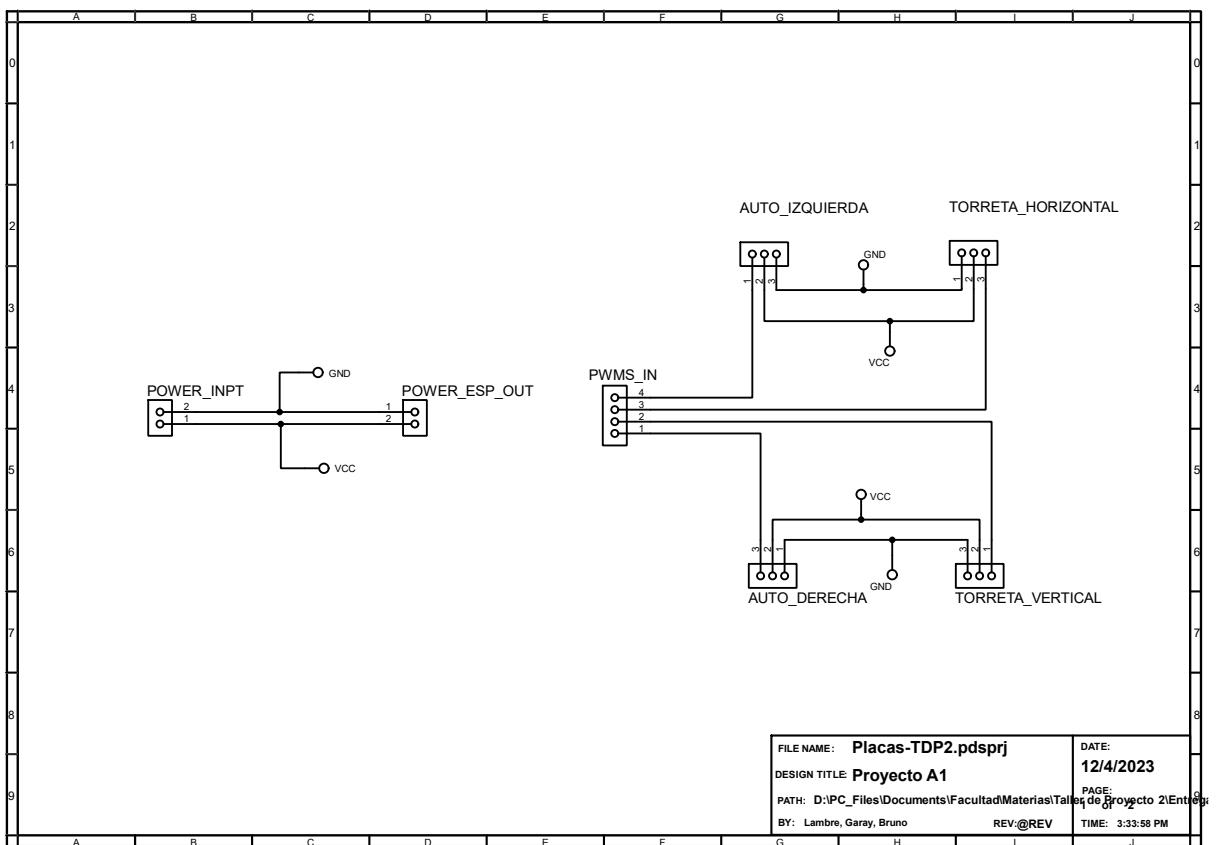


Figura 2 – Diagrama esquemático de la PCB montada sobre el vehículo

Cada componente consiste en:

- POWER\_INPT: conexión de la fuente de alimentación con el proyecto, la cual se utiliza para alimentar los servomotores y el resto de los componentes.
- POWER\_ESP\_OUT: alimentación para las ESP32-CAM que se encuentran en la otra PCB desarrollada.
- PWMS\_IN: recepción de los PWM generados por los ESP32-CAM para mover los servomotores.

- AUTO\_IZQUIERDA: conexión del servomotor de 360 grados para el control del movimiento de la rueda izquierda del vehículo.
- AUTO\_DERECHA: conexión del servomotor de 360 grados para el control del movimiento de la rueda derecha del vehículo.
- TORRETA\_VERTICAL: conexión del servomotor para el control del movimiento vertical de la torreta.
- TORRETA\_HORIZONTAL: conexión del servomotor para el control del movimiento horizontal de la torreta.

Una vez desarrollado el esquemático, se procedió a la creación de una PCB. Se debió tener en cuenta el espacio entre la placa y la torreta para evitar el enredo de los cables cuando se mueva. El diagrama final de la PCB es el siguiente:

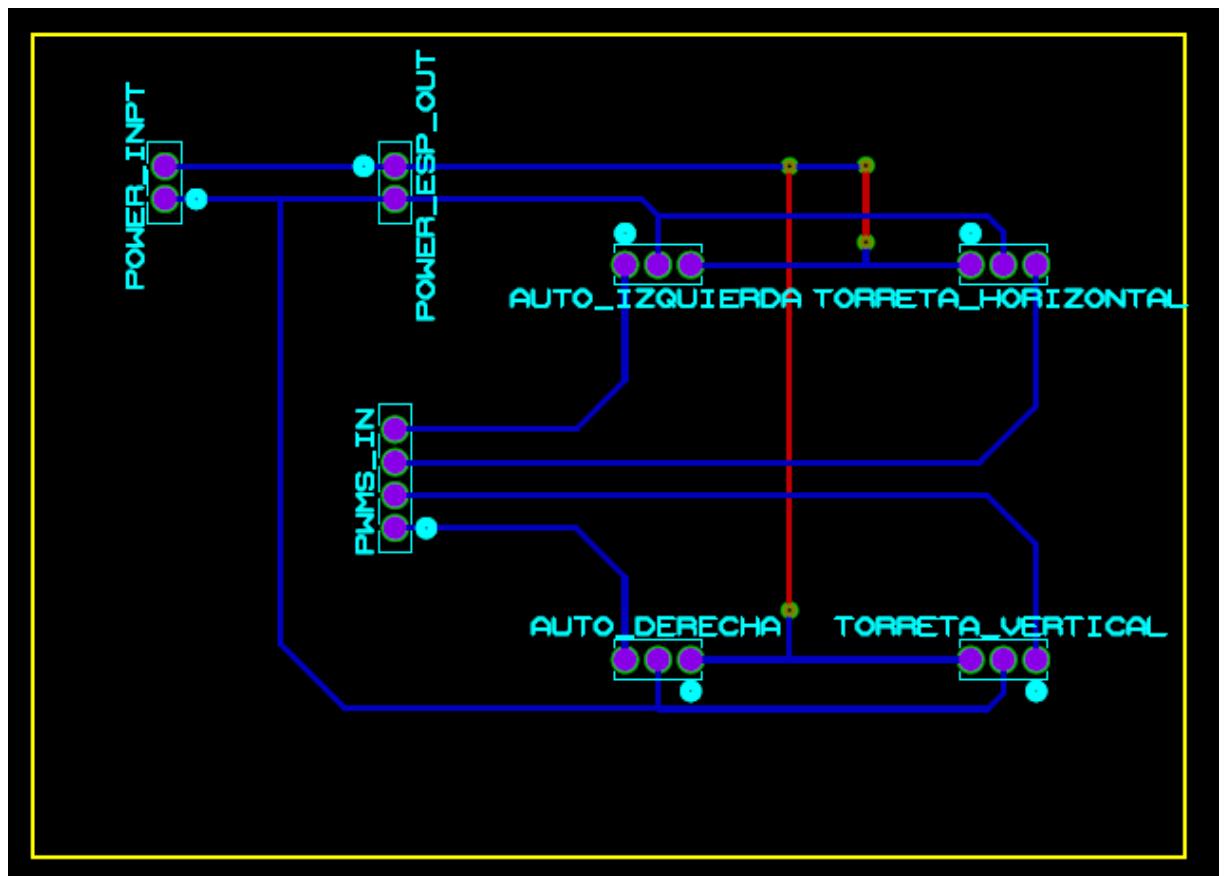
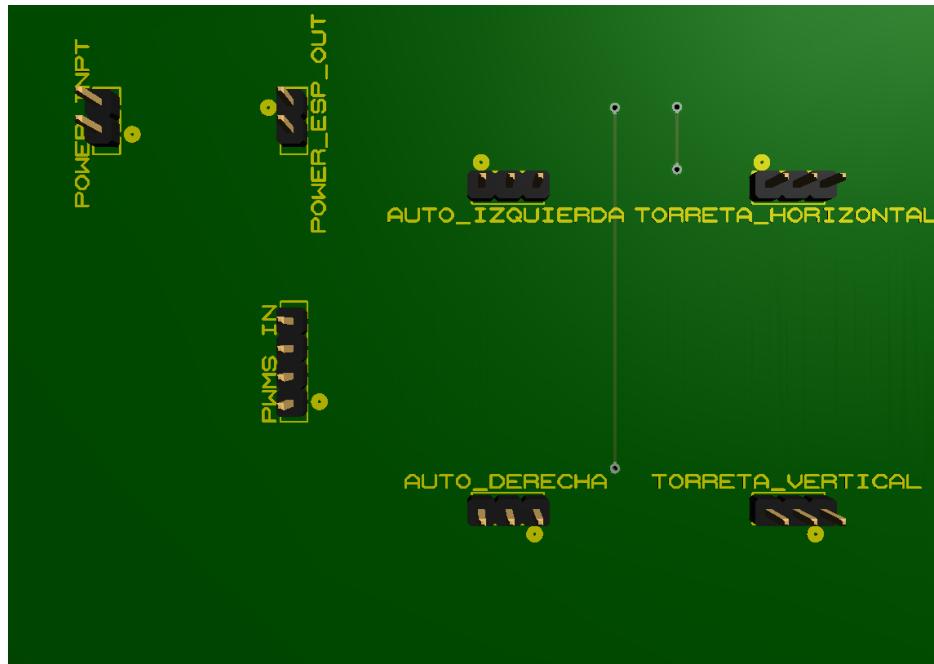
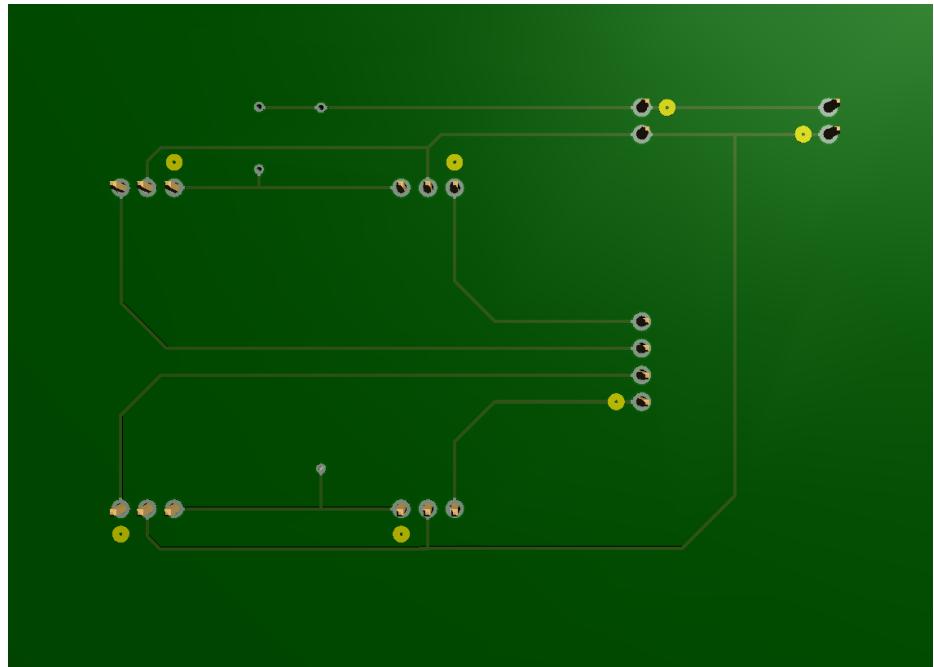


Figura 3 – Diagrama de la PCB diseñada

Vista desde una perspectiva en 3D, la PCB se ve de la siguiente manera:



**Figura 4** – Modelo 3D de la PCB - Frente



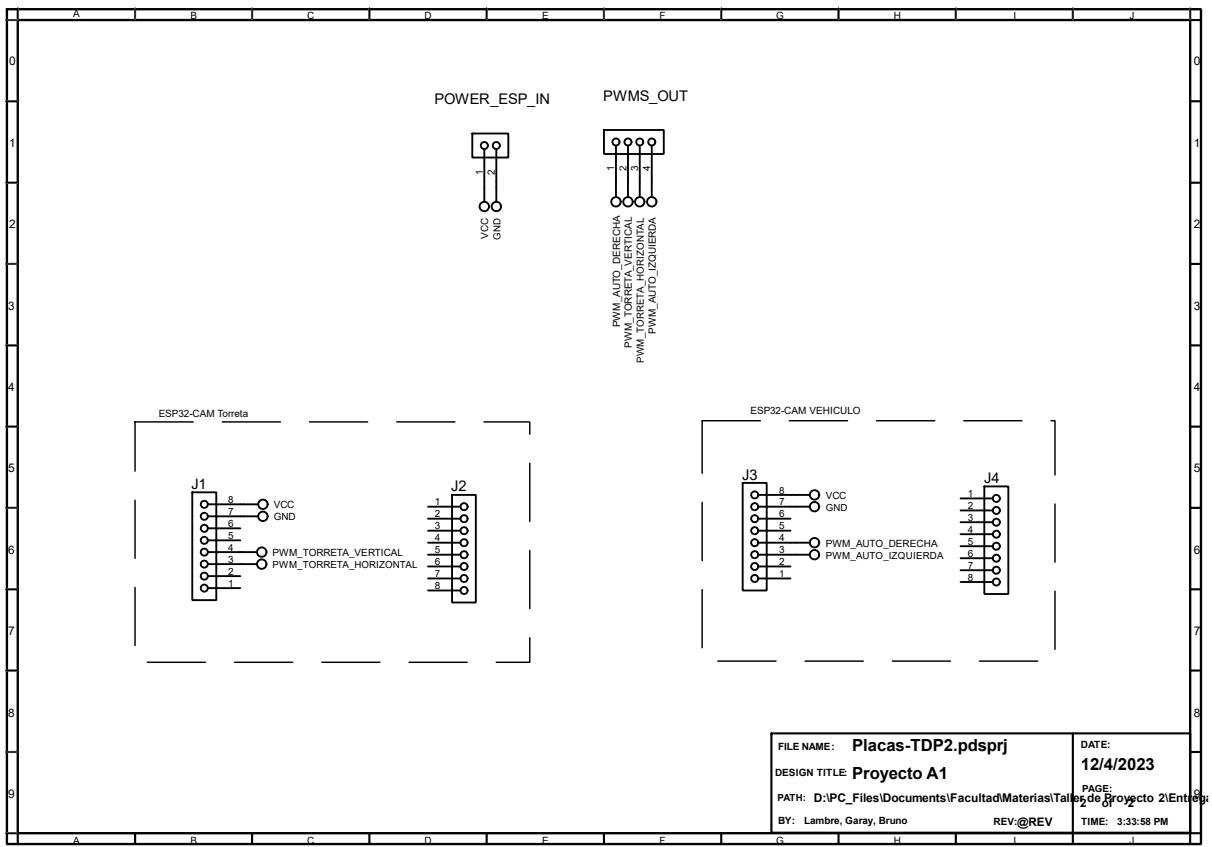
**Figura 5** – Modelo 3D de la PCB – Revés

Para información sobre la construcción física, ver sección 4.1.1

### 2.1.2. Placa Torreta

Esta placa aloja dos módulos ESP32. Su función principal es la de contener los ESP32-CAM y enviar las señales PWM a la “Placa Auto” para el control de los servomotores. También recibe alimentación de la otra placa.

El diagrama esquemático es el siguiente:



**Figura 6** – Diagrama esquemático de la PCB montada sobre la torreta

Cada componente consiste en:

- **POWER\_ESP\_IN:** Conexión de entrada de la alimentación para esta placa. Tiene como objetivo alimentar las ESP32-CAM.
- **PWMS\_OUT:** salida de los PWM de las ESP hacia la “PCB AUTO” para mover correctamente los servomotores.
- **J1 y J2:** Ambos componentes sujetan la ESP encargada del movimiento de la torreta.
- **J3 y J4:** Ambos componentes sujetan la ESP encargada del movimiento del vehículo.

Una vez desarrollado el esquemático, se procedió a la creación de una PCB. Se debió tener en cuenta la distancia entre los conectores de 1 misma ESP para que la misma entre correctamente. También, se buscó que las ESP estén lo más cerca posible para que las imágenes se superpongan correctamente en las gafas VR. El diagrama final de la PCB es el siguiente:

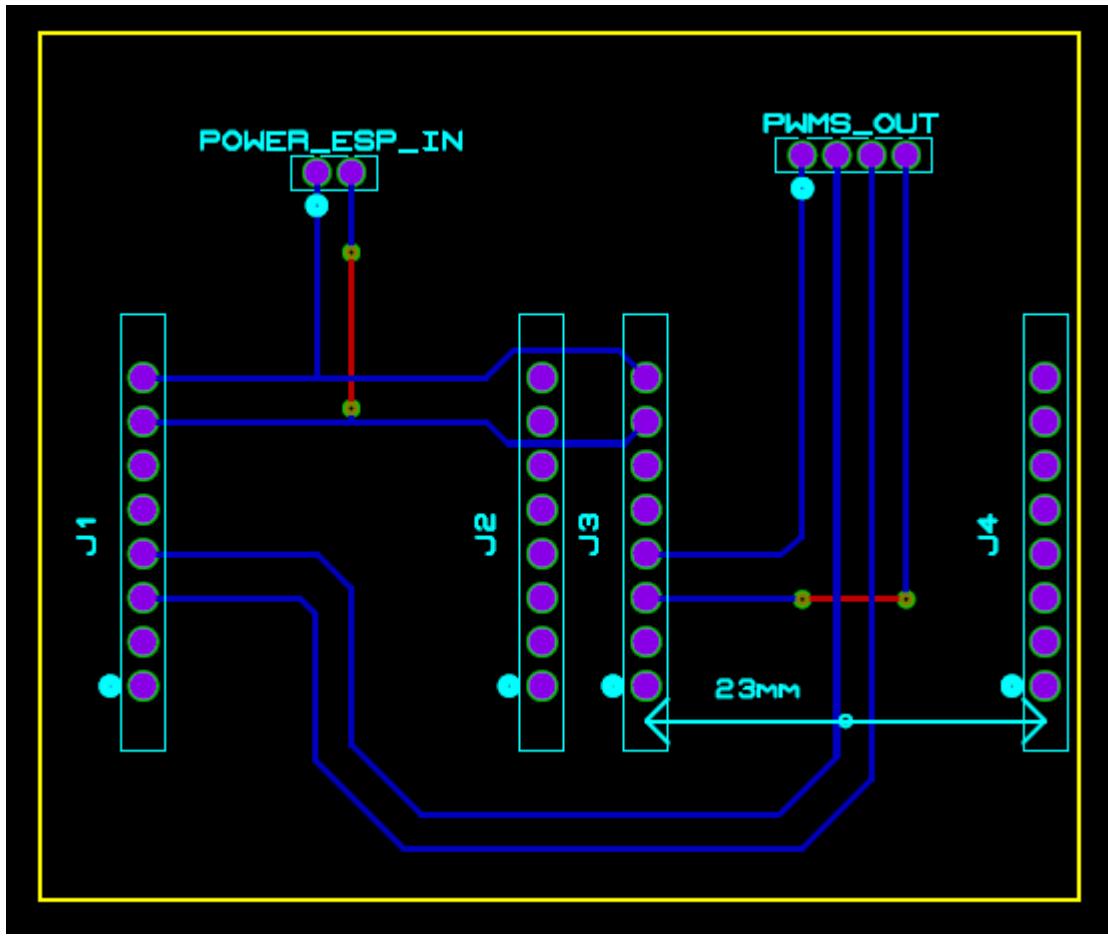


Figura 7 – Diagrama de la PCB diseñada

Vista desde una perspectiva en 3D, la PCB se ve de la siguiente manera:

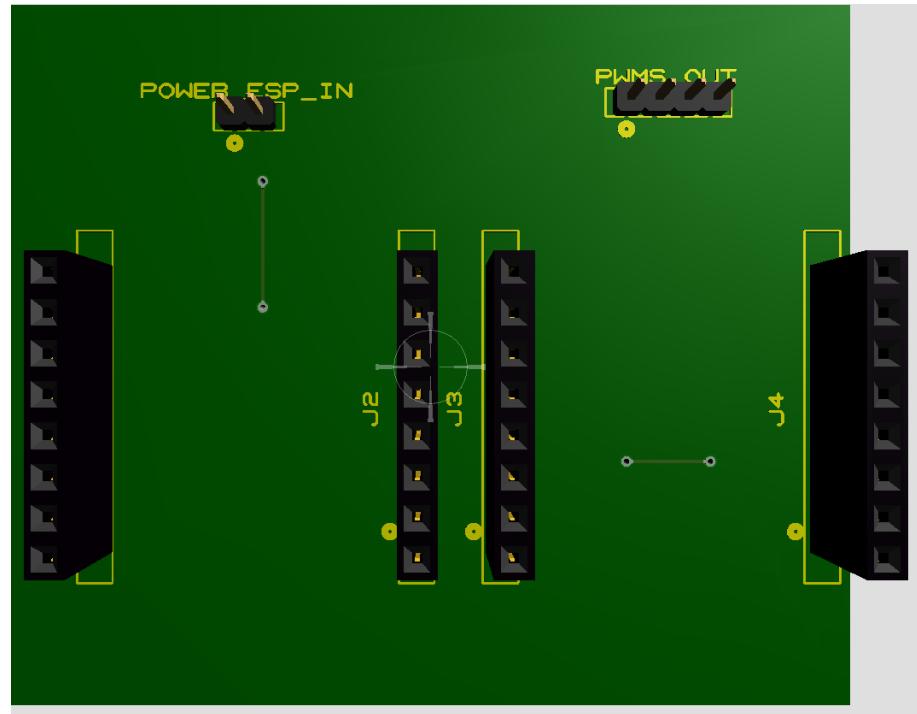
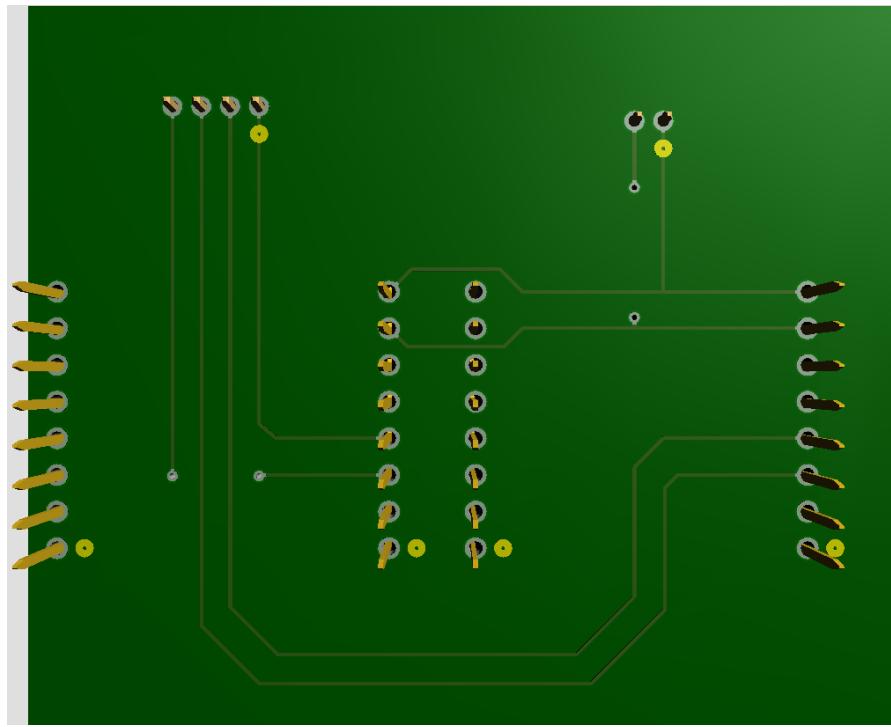


Figura 8 – Modelo 3D de la PCB – Frente



**Figura 9** – Modelo 3D de la PCB – Revés

Para información sobre la construcción física, ver sección 4.1.1

### 3. Documentación del Software del Proyecto

#### 3.1. Bloque de gafas VR y giroscopio

Teniendo en cuenta el objetivo de que la torreta mencionada previamente rote en dos ejes basándose en la posición de la cabeza del usuario, uno de los procesos de esta primera etapa fue el de comenzar a trabajar con el giroscopio (módulo MPU6050) y observar cómo lograríamos que esos datos fueran enviados a una placa de desarrollo con conectividad Wi-Fi para que, finalmente, ésta se comunicara con la torreta.

En base a esto, se procederá a profundizar dentro de cada etapa del desarrollo de este bloque para esclarecer la implementación y funcionamiento de cada una.

##### 3.1.1. Estudio del módulo MPU6050

Antes de iniciar cualquier prueba, estudiamos la hoja de datos proporcionada por el fabricante del módulo (consultar Bibliografía). A partir de esto, observamos dos cuestiones de importancia: la tensión de alimentación y los datos capturados por el módulo.

En cuanto a la alimentación, en la Figura 10, observamos que la tensión óptima para este componente oscila alrededor de 3.3V. Por lo tanto, decidimos utilizar uno de los pines de la placa de desarrollo (ESP32) que proporciona esta tensión.

<b>Part / Item</b>	<b>MPU-6000</b>	<b>MPU-6050</b>
<b>VDD</b>	2.375V-3.46V	2.375V-3.46V
<b>VLOGIC</b>	n/a	1.71V to VDD
<b>Serial Interfaces Supported</b>	I <sup>2</sup> C, SPI	I <sup>2</sup> C
<b>Pin 8</b>	/CS	VLOGIC
<b>Pin 9</b>	AD0/SDO	AD0
<b>Pin 23</b>	SCL/SCLK	SCL
<b>Pin 24</b>	SDA/SDI	SDA

**Figura 10** – Especificación de pines MPU6050

Por otro lado, en lo que respecta a los datos que podemos recibir de este módulo, notamos que además de contar con giroscopios (uno para cada una de las direcciones x, y, z), también posee acelerómetros y un termómetro para medir la temperatura ambiente. Sin embargo, inicialmente, esta información no es relevante para el proyecto. Lo que resultó de especial importancia es que los datos entregados por los giroscopios no están expresados en una magnitud de rotación (como grados o radianes), sino en términos de aceleración angular ( $\omega$ ). A partir de esto, definimos el procesamiento que la placa deberá realizar para convertir estos datos a grados, de manera que podamos enviar la información necesaria a los servos para lograr una rotación adecuada. En las secciones siguientes profundizaremos en este último tema.

### 3.1.2. Configuración del entorno para trabajar con la placa de desarrollo

Para trabajar con la placa de desarrollo proporcionada por la cátedra (ESP32s) se investigó la forma de instalar sus drivers en las computadoras utilizadas para el desarrollo del proyecto y las librerías necesarias para poder programar la misma en el entorno del Arduino IDE (dados que el equipo cuenta con experiencia previa utilizando esta herramienta).

Finalmente, gracias al aporte de [esta](#) entrada a un blog, logramos obtener los recursos precisados. A su vez, mediante pruebas concluimos en que la placa de desarrollo a seleccionar en el IDE resulta ser la “ESP32-WROOM-DA Module”.

A continuación, se puede encontrar una imagen representativa de la placa de desarrollo utilizada para esta parte del proyecto.



**Figura 11** - Placa ESP32s

### 3.1.3. Programación de la placa y el giroscopio

Una vez establecida la configuración del entorno de programación, el primer paso para poder recibir los datos del giroscopio consistió en descargar e importar las librerías necesarias. A partir de esto, y de realizar las conexiones pertinentes (ver Figura 12), es que pudimos obtener las primeras lecturas de los datos del giroscopio.

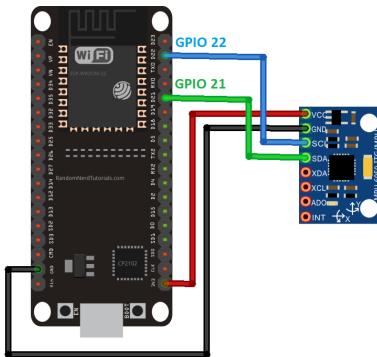


Figura 12 - Conexión ESP – MPU

Como se mencionó anteriormente, los datos obtenidos por el giroscopio refieren a una aceleración angular, que responde a la siguiente ecuación:

$$\omega = \frac{\theta}{dt}$$

Siendo  $\theta$  el ángulo de rotación expresado en grados y  $dt$  el tiempo transcurrido entre la medición actual y la anterior. Es entonces, a partir de esta expresión, que logramos obtener el dato de rotación en grados dependiendo de la frecuencia de muestreo. Por lo tanto, si la medición se hiciera cada 0.5 segundos, la expresión resultaría:

$$\theta = \omega dt$$

Una vez obtenido el dato de cuántos grados se movió el sensor desde la última medición se suma a una variable en la que se almacena la rotación absoluta, es decir, la cantidad de grados a los que se encuentra el sensor de su origen, ya que es esta información la que deben recibir, finalmente, los servos.

$$rotación = rotación + \theta$$

Finalmente, esta metodología permitió un seguimiento preciso y confiable de la orientación del sensor, esencial para el control efectivo de la torreta. Así, establecimos un puente sólido entre la detección de movimientos y su correspondiente respuesta mecánica, sentando las bases para las siguientes fases del proyecto.

### 3.1.4. Modificación del cálculo de rotación

Luego de varios ensayos y pruebas, en la búsqueda del causante del desfasaje obtenido en la medición de la rotación cuando el movimiento del módulo era realizado con cierta velocidad se llegó a la conclusión de que, dado que en la versión hasta ahora presentada del código la medición de los eventos del MPU6050 se está realizando con una espera de 50ms entre ellas, es posible que se estén perdiendo algunos movimientos que el usuario realiza con el giroscopio durante ese intervalo de tiempo. Es por esto que se evaluaron dos posibilidades para mejorar este cálculo.

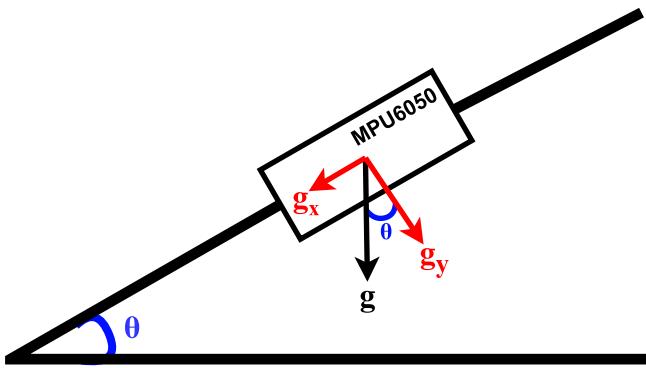
En primer lugar, se propuso achicar el tiempo entre mediciones del giroscopio, de manera que se reduzca la probabilidad de perder movimientos realizados por el usuario; siguiendo esta lógica, para obtener un resultado exacto de la rotación, el tiempo de muestreo debería aproximarse a 0. De esta forma, el cálculo del diferencial de rotación en cada instante respondería a la siguiente fórmula.

$$\lim_{dt \rightarrow 0} \omega(x + dt) - \omega(x)$$

Siendo  $\omega(x)$  la aceleración angular que se obtiene del giroscopio en un momento  $x$  dado, y  $dt$  el tiempo que transcurre entre cada medición.

De esta forma, se podría sumar este diferencial a la variable que almacena la rotación absoluta. Sin embargo, esta solución implicaría una medición constante por parte del sensor y un tiempo infinitesimalmente pequeño para las otras instrucciones del programa, lo cual es impracticable en un entorno real debido a las limitaciones de tiempo de ejecución y recursos del sistema.

Es por esto por lo que se optó por realizar el cálculo de la rotación de otra manera, esta vez aprovechando la información proporcionada por los acelerómetros. Para realizar este cálculo, se parte de la base de que, para la aplicación en desarrollo, la única fuerza que se estará realizando sobre el módulo será la gravedad, por lo que, las mediciones obtenidas por los acelerómetros en cada eje, responderán a esta fuerza. Con esta premisa, el diagrama de fuerzas se verá de la siguiente manera (ver Figura 13).



**Figura 13 – Diagrama de fuerzas sobre el MPU**

El diagrama previamente enseñado es una simplificación en dos dimensiones de las fuerzas que aplican sobre el módulo, siendo  $g$  la fuerza de la gravedad con sus componentes en cada eje  $y$   $\theta$  el ángulo de inclinación, en este caso, con respecto al eje  $x$ .

A partir de esto, se llega a la siguiente expresión para obtener la inclinación:

$$\theta = \tan^{-1}\left(\frac{g_x}{g_y}\right)$$

De esta forma, la expresión equivalente para tres dimensiones resulta:

$$\theta_x = \tan^{-1}\left(\frac{g_x}{\sqrt{(g_y)^2 + (g_z)^2}}\right)$$

A partir de la implementación de este cálculo, el valor de la rotación en cada eje del módulo ya no presentó desfasajes distintos para la misma posición, aunque sí uno fijo, que responde a la siguiente fórmula:

$$\text{rotación} = \theta_n + k$$

Siendo  $\theta$  el ángulo real y  $k$  un valor del error de medición, constante para todos los instantes. Es por esto que, en la sección próxima se explicitará la manera de calcular este error para corregirlo posteriormente.

### 3.1.5. Calibración del MPU6050

Después de investigar diversas fuentes y estudiar las librerías implementadas para las placas de desarrollo Arduino, que permiten calibrar la medición de los eventos del sensor, diseñamos un método para obtener el valor del error de medición de la rotación en cada eje y restarlo a la rotación obtenida. Este enfoque implica, al inicio del programa y con el sensor en reposo, tomar 100 muestras del ángulo obtenido para el eje deseado. Dado que el sensor se encuentra inmóvil en este estado, cualquier valor diferente de cero se considera un error.

Después de recolectar las 100 muestras, se calcula un promedio de estos valores, y ese número se resta en cada medición del sensor

Gracias a esto, se puede solucionar un error considerable en la medición, en el orden de los 2 a 3 grados, lo que podría haber afectado el producto final. En la Figura 14 se puede observar un ejemplo del resultado de la ejecución del programa de calibración.

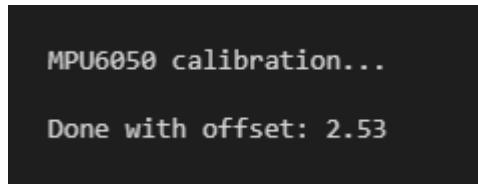
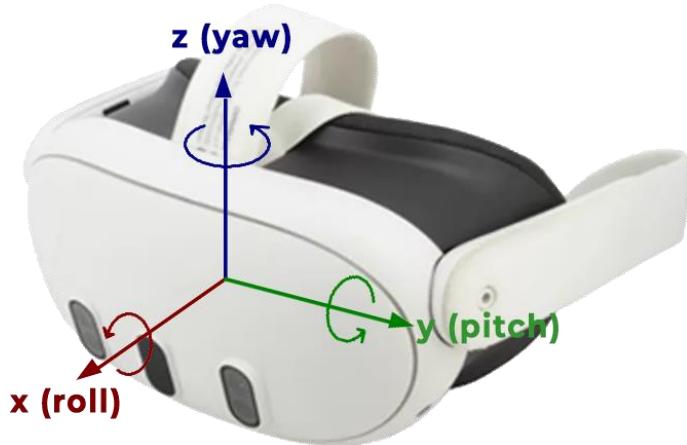


Figura 14 – Resultado del programa de calibración

### 3.1.6. Medición de rotación con respecto al eje Z

Llegados a este punto, se han realizado experimentos para medir la rotación en relación con los ejes  $x$  e  $y$ , los cuales han demostrado resultados positivos. Sin embargo, en la implementación final, los ejes relevantes para el movimiento de la torreta son el  $y$  y el  $z$ , los cuales corresponden a los movimientos conocidos como “pitch” y “yaw”, respectivamente (véase la Figura 15 para mayor detalle).

Consecuentemente, se iniciaron experimentos para medir la rotación alrededor del eje  $z$ , sin embargo, dichas pruebas revelaron anomalías significativas en los datos proporcionados por el módulo de medición. Esto se debe a que los acelerómetros del MPU son extremadamente eficaces para medir el “pitch” y el “roll”, gracias a la presencia de la fuerza de gravedad, que actúa como un vector constante y predecible hacia abajo, sirviendo de referencia para el sensor. Al modificar la orientación del dispositivo, la dirección de la gravedad relativa al mismo se altera, lo cual es detectado por el acelerómetro. No obstante, el movimiento de “yaw” implica una rotación alrededor del eje vertical, análogo a girar una brújula sobre una mesa. En esta modalidad de movimiento, la dirección de la fuerza gravitatoria relativa al dispositivo permanece inalterada, dado que la gravedad continúa actuando en la misma dirección, independientemente de la rotación alrededor de este eje vertical. Es por esto que el MPU6050 no provee datos útiles para la medición del “yaw”.



**Figura 15** – Rotación sobre los ejes del espacio

Para solucionar esto, dado que el proyecto carece de un sensor dedicado a esta medición, únicamente para la medición de este tipo de rotación, se decidió volver al método anterior (sección 3.1.3.), el cual, si bien presenta fallos debido a su naturaleza acumulativa, provee una medición medianamente acertada para este tipo de rotación.

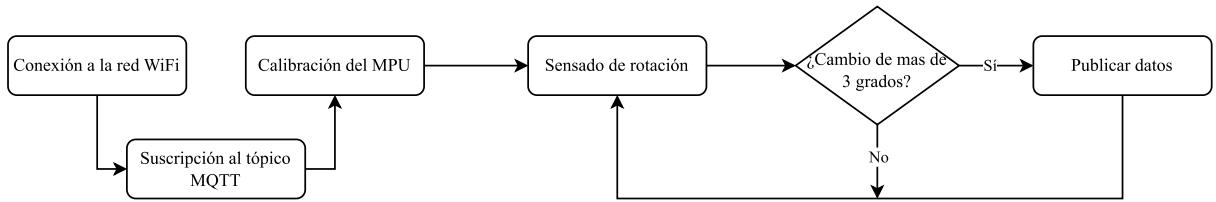
### 3.1.7. Control remoto de la torreta con MPU6050

Una vez que se obtuvieron las mediciones de la rotación en los dos ejes deseados, se prosiguió con el envío de estos datos mediante un protocolo de comunicación inalámbrica, de manera que, mientras el usuario usa las gafas con el giroscopio, la torreta con los servos se encuentre montada en el vehículo y se controle con los movimientos cervicales del usuario.

Por motivos que se detallarán en secciones posteriores de este informe, el método de comunicación elegido fue mediante un servidor MQTT central, al cual la placa del MPU se conecta mediante WiFi para enviarle los datos previamente mencionados a través de un tópico específico. De esta forma, el trabajo de la placa ESP32s se centra en dos tareas principales: recoger y procesar los datos del MPU6050 y comunicar estos datos al servidor MQTT. Para lograr esto, se utilizó la librería WiFi del Arduino IDE para establecer una conexión con la red y la librería PubSubClient para interactuar con el servidor MQTT.

Por último, se agregó una sentencia en el código para que los datos de la rotación sean enviados únicamente si difieren en 3 grados o más con respecto a la última vez que se mandaron, de esta forma, se evita sobrecargar la red con mensajes innecesarios y se asegura un poco más de estabilidad en el movimiento.

Habiendo mencionado esto, se podrá observar a continuación un diagrama de flujo del código cargado en la placa ESP32s con el MPU6050.



**Figura 16** – Funcionamiento del código ESP-MPU

Como se puede observar en la Figura 16 el código, primero se establece la conexión WiFi de la ESP32s con la red y luego se configura la conexión al servidor MQTT. Una vez conectada, la placa recoge los datos de rotación del MPU6050, ya calibrados, y, en caso de presentar cambios sustanciales, los envía al servidor MQTT en un formato predefinido. El servidor, a su vez, redirigirá estos datos a los servos de la torreta montada en el vehículo.

### 3.2. Servidor central

Uno de los bloques principales de nuestro proyecto es el servidor donde se concentran la mayoría de las comunicaciones. A diferencia de los demás microcontroladores que usamos en el proyecto, este es una máquina de propósito general, donde estarán ejecutándose dos servidores fundamentales para el funcionamiento del sistema: un servidor web al que se accederá desde un dispositivo móvil con pantalla y las capacidades para acceder a una página web con el propósito visualizar las cámaras mediante las gafas VR; y un servidor MQTT donde se ubicará el bróker de MQTT para la comunicación entre varias partes del sistema.

En este bloque del proyecto, se logró desplegar una máquina virtual en un entorno aislado y controlado con Debian 12 y un bróker de Mosquitto. Se espera que dicha máquina virtual encapsule toda la funcionalidad que requiera de librerías y/o instalaciones específicas de modo que si se desea desplegar el proyecto en otro ambiente, migrarlo a un entorno de producción, o restaurarlo en caso de fallas, sólo sea necesario importar la imagen del sistema.

En cuanto al sistema operativo, se eligió Debian 12 debido a su estabilidad, seguridad y soporte a largo plazo, lo que lo hace ideal para servidores que requieren un funcionamiento ininterrumpido. Además, Debian tiene una amplia base de usuarios y una gran cantidad de documentación disponible, lo que facilita la resolución de problemas y brinda soporte.

Respecto a la instalación, solo fueron instaladas las características mínimas para su funcionamiento, lo que significa que no hay software superfluo que pueda presentar riesgos de seguridad o consumir recursos del sistema innecesariamente. Esto es especialmente útil en un entorno de servidor donde los recursos y la seguridad son de suma importancia.

### **3.3. Servidor Web**

Uno de los principales objetivos de este proyecto consiste en la transmisión del streaming de video a un servidor web, del cual se accederá desde un celular que se colocará dentro de las gafas VR para tener una experiencia inmersiva al controlar el auto. El servidor Web se ejecutará dentro del servidor central (Ver sección 4.3 “Servidor Central”).

#### **3.3.1. Flask Web Server con dos ESP32-CAM diseñado para las gafas VR**

Luego de las pruebas iniciales, se optó por utilizar la librería de Python “Flask” para el desarrollo del servidor web, dado que su implementación es sencilla, y sus capacidades son acordes a las necesidades del proyecto: una única página donde se vea el video de ambas ESP32-CAM de forma que resulte óptimo para su visualización desde unas gafas VR.

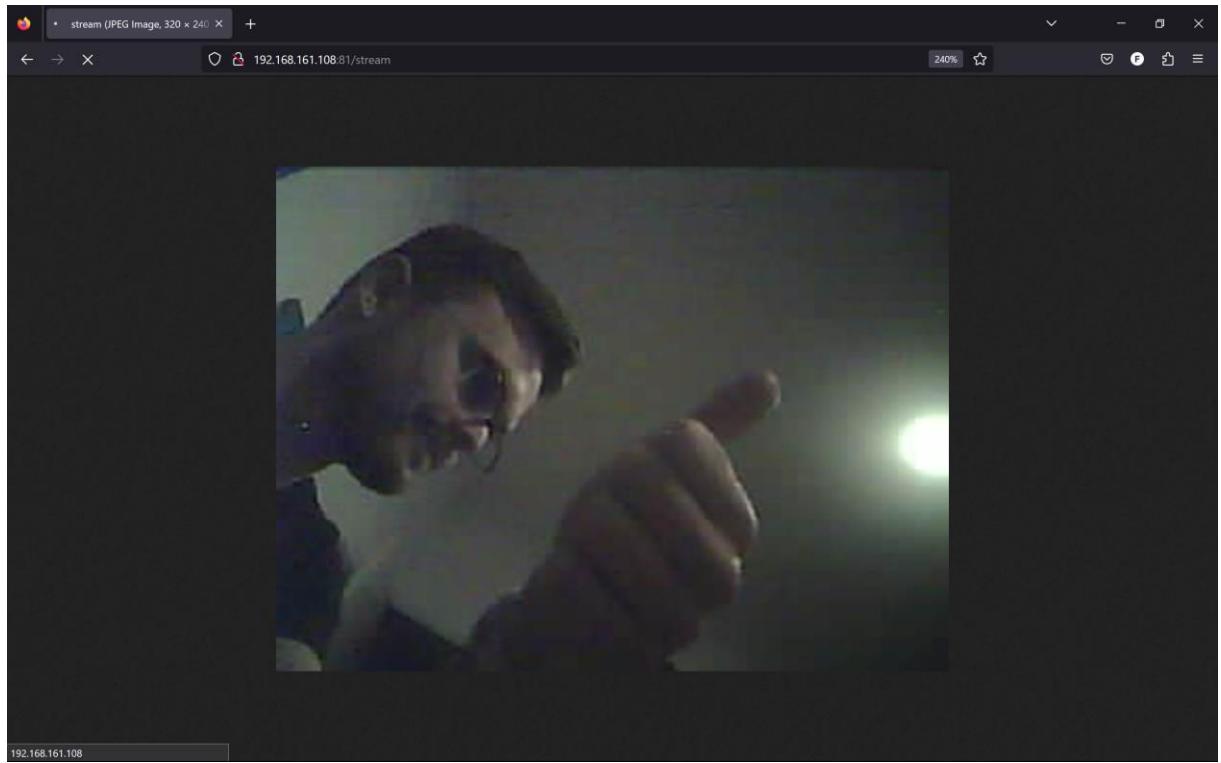
Prueba de lo sencillo que es el uso de Flask es el contenido del archivo “app.py”, donde se configura al Web Server, en el Apéndice C “Configuración del Flask Web Server” se puede ver el código de dicha configuración.

Lo único a resaltar en dicho código es la variable app, en la cual se almacena una instancia de la clase Flask. Dicha instancia contendrá el comportamiento y los atributos de nuestra aplicación Web. Luego, con el método route('/') se define la ruta inicial de la aplicación y la función a ejecutar, en este caso, la función “index” de Python renderiza el archivo “index.html”. En dicho archivo son de importancia las siguientes dos líneas:

```


```

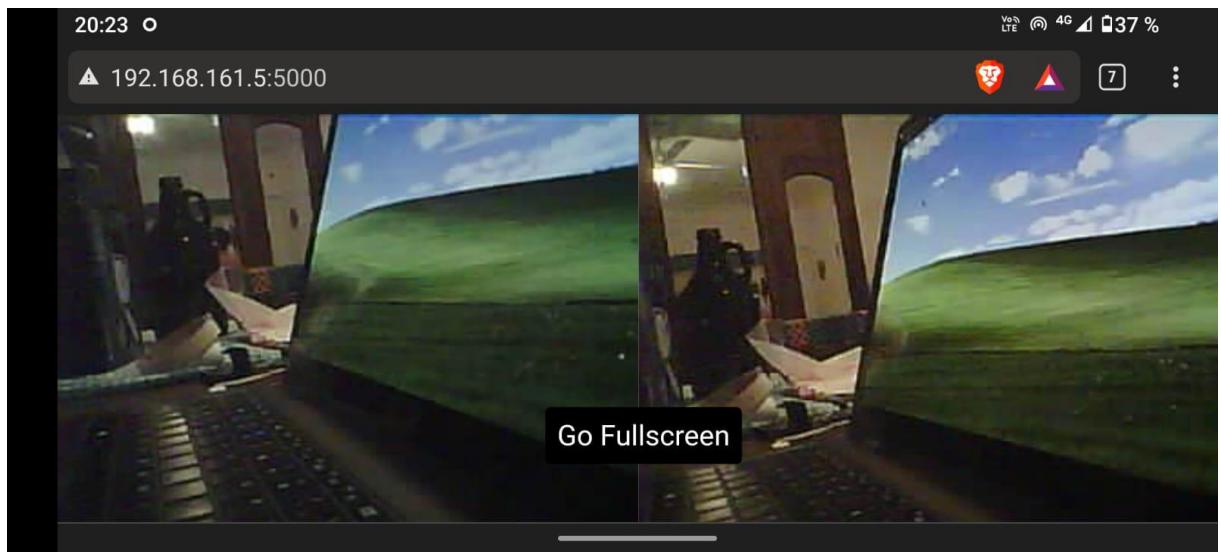
Aquí, usando el tag “img” de HTML, se obtiene el video de las dos ESP32-CAM, ambos en el puerto 81 y el recurso /stream. Si entramos a alguno de esas direcciones veremos el siguiente stream (Figura 17).



**Figura 17** – Stream de una ESP32-CAM visto desde un navegador Web.

Además, con el tag “button”, se agregó un botón que permite al usuario acceder al modo pantalla completa.

El sitio web final se puede ver en la Figura 18.



**Figura 18** – Página principal (y única) del Web Server desplegado con Flask

Si se presiona el botón “Go Fullscreen” los streams de video se pueden ver tal como en la Figura 19.



**Figura 19** - Página principal (y única) del Web Server desplegado con Flask en pantalla completa

En [este link](#) se puede ver un video donde se muestra el funcionamiento del Web Server.

### 3.3.2. Análisis de las estadísticas de la transmisión de video utilizando TCP

Uno de los aspectos importantes acerca del funcionamiento del sistema es la transmisión de video. Una gran parte de este proyecto se dedicó al análisis e investigación de posibles protocolos tanto de capa de aplicación como de transporte. Finalmente, la decisión final fue utilizar HTTP en capa de aplicación y TCP en capa de transporte. La razón de esta elección se debe a la funcionalidad obtenida con dichos protocolos, sumado al soporte que hay para los mismos en las ESP32CAM. En determinados momentos del desarrollo se planteó utilizar RTSP en capa de aplicación y UDP en capa de transporte, aunque finalmente fueron descartados, para obtener más detalles acerca de ese análisis ver el Apéndice B, “Protocolo RTSP sobre UDP”.

Una vez tomada la decisión de los protocolos a utilizar y comprobada su funcionalidad, se decidió que era pertinente realizar medidas acerca de las capacidades de los mismos. Para esto se utilizó la funcionalidad que permite Wireshark para analizar conversaciones entre nodos de la red, se puede ver en la Figura 20 estadísticas acerca de la cantidad de paquetes que son transmitidos y la cantidad recibidos entre la ESP32CAM (IP: 192.168.137.222) y el Web Server (IP: 192.168.137.1) en un período de cinco minutos.

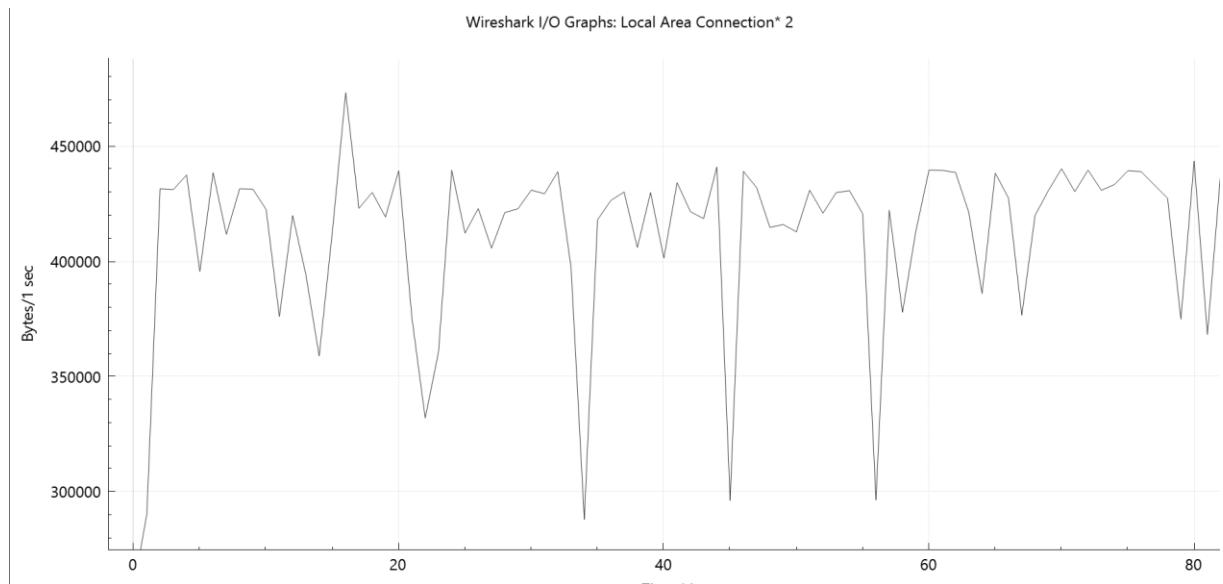
Address	Port	Packets	Bytes	Total Packets	Percent Filtered	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes
192.168.137.1	32319	106,152	92 MB	106,152	100.00%	33,419	2 MB	72,733	90 MB
192.168.137.222	81	106,152	92 MB	106,152	100.00%	72,733	90 MB	33,419	2 MB

**Figura 20** - Estadísticas de paquetes entre una ESP32CAM y el Web Server

Lo principal que llama la atención de la Figura 20 es que todos los paquetes enviados en la conversación llegar a su receptor sin problema alguno. Además se puede notar cómo la

ESP32CAM envía muchos más datos (en MB) que el WebServer debido a la transmisión de video.

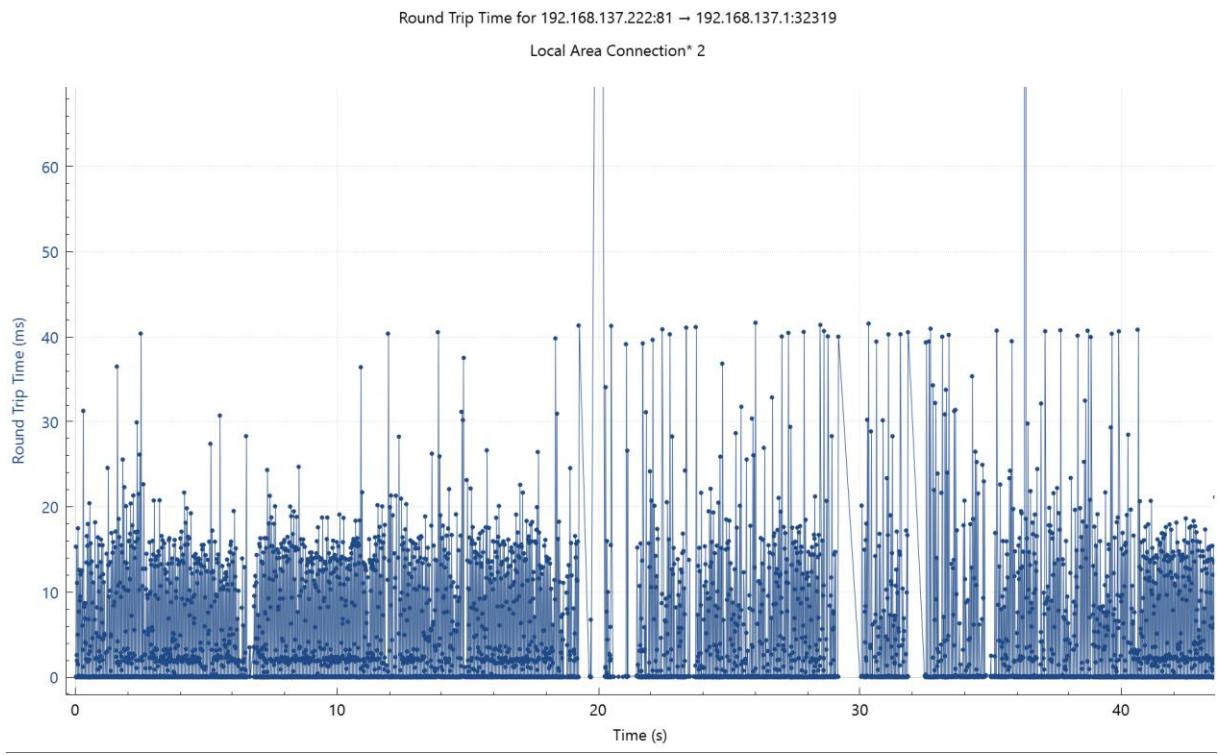
El siguiente gráfico que observaremos (Figura 21), resulta interesante ya que permite analizar cuántos bytes por segundo se pueden enviar entre los dos nodos ESP32CAM y Web Server.



**Figura 31** – Cantidad de Bytes por segundo desde la ESP32-CAM al Web Server a lo largo del tiempo

Como se puede ver, se transmiten en promedio 410 kB por segundo desde las ESP32CAM, lo que resulta en un video de unos 15 FPS, lo que cumple con los requerimientos funcionales del proyecto.

Finalmente, analizaremos el Round Trip Time (RTT) entre la ESP32CAM y el WebServer. El que mide el tiempo que tardan los paquetes en viajar desde el emisor al receptor y viceversa, impacta directamente en la velocidad a la que se pueden enviar datos y en la estabilidad de la conexión. Un RTT alto puede indicar congestiones en la red o problemas de ruta, llevando a retrasos, buffering, y una calidad de video reducida. En nuestra red, podemos ver en la Figura 22 que el RTT no supera normalmente lo 45ms, lo cual resulta más que aceptable para la aplicación. Hay algunos picos de 100ms, pero no afectan significativamente el funcionamiento del sistema.



**Figura 42 - RTT entre una ESP32CAM y el Web Server**

### 3.4. Servidor MQTT

El servidor MQTT se encuentra ubicado dentro del servidor central (Ver sección “Servidor Central”) y se utilizará el bróker de MQTT, Mosquitto, tal como se puede ver en la Figura 23.

```
Applications Xfce Terminal 36% - 1:52 2023-10-05 19:15 admin-autito
Terminal - admin-autito@proyecto-autito: ~
File Edit View Terminal Tabs Help
admin-autito@proyecto-autito:~$ systemctl status mosquitto
● mosquitto.service - Mosquitto MQTT Broker
  Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; preset: )
  Active: active (running) since Thu 2023-10-05 14:16:21 EDT; 34s ago
    Docs: man:mosquitto.conf(5)
           man:mosquitto(8)
   Process: 494 ExecStartPre=/bin/mkdir -m 740 -p /var/log/mosquitto (code=exited, pid=494)
   Process: 497 ExecStartPre=/bin/chown mosquitto /var/log/mosquitto (code=exited, pid=497)
   Process: 504 ExecStartPre=/bin/mkdir -m 740 -p /run/mosquitto (code=exited, pid=504)
   Process: 510 ExecStartPre=/bin/chown mosquitto /run/mosquitto (code=exited, pid=510)
 Main PID: 513 (mosquitto)
   Tasks: 1 (limit: 4712)
  Memory: 1.7M
     CPU: 62ms
    CGroup: /system.slice/mosquitto.service
              └─513 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Warning: some journal files were not opened due to insufficient permissions.
lines 1-17/17 (END)
```

**Figura 23 - Status de mosquitto en Debian 12**

El sistema está de modo que Mosquitto se ejecute automáticamente cada vez que se prende la máquina virtual, además, se lo configuro para que permita conexiones al puerto de MQTT, 1883, desde cualquier host a cualquiera de sus interfaces.

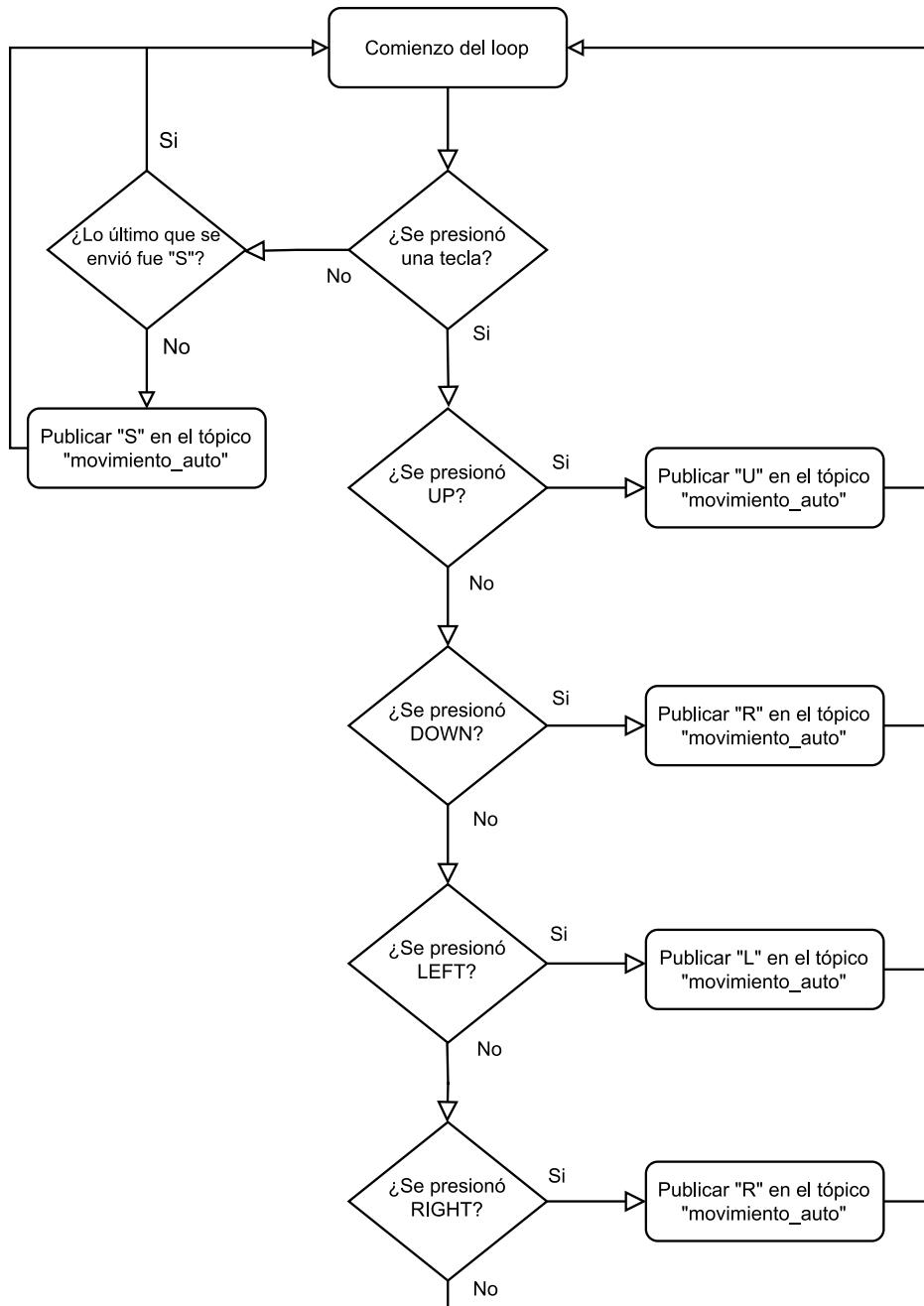
Como ya se mencionó, en dicha máquina virtual se concentra la mayoría de las comunicaciones, puesto que el broker MQTT será el encargado de gestionar dos “tópicos” principales:

- Tópico donde se publique la posición del giroscopio para el movimiento de la torreta: aquí el publicador será el NodeMCU, mientras que el subscriptor principal será la ESP32-CAM encargada del movimiento de la torreta.
- Tópico “auto” donde se publican las teclas presionadas: aquí el publicador será en principio el servidor central, aunque en futuros desarrollos podrían ser dispositivos aparte, y el subscriptor principal será la ESP32-CAM encargada del movimiento del auto.

### **3.4.1. Publicador MQTT a ESP32-CAM que controla el auto**

Para el envío de las teclas presionadas a la ESP32CAM que controla el movimiento del auto, se utiliza la aplicación que se encuentra en el archivo “Publisher.py”. En la misma, un publicador MQTT envía una string: “U”, “D”, “L”, o “R” según la tecla presionada. Esto se realizó usando el lenguaje de programación “Python” y las librerías “paho.mqtt.client” para el cliente MQTT, y “curses” para la detección de teclas. Para más detalles acerca de la implementación ver el Apéndice C “Código del publicador MQTT al auto”.

En la implementación se almacena en una variable “c” la tecla presionada por el usuario. La lógica una vez obtenida la tecla presionada por el usuario es la que se puede ver en el siguiente diagrama de flujo (Figura 24).



**Figura 24** – Diagrama de flujo del publicador a la ESP32-CAM que controla el auto

En este diagrama de flujo se muestra el procesamiento que se hace en el servidor central dentro del programa que actúa como “publicador” de MQTT. En el mismo se observa cómo, según la flecha del teclado que el usuario presione (UP, DOWN, LEFT o RIGHT) se publicarán los correspondientes comandos en el tópico “auto”, donde la ESP32-CAM estará como suscriptora.

Otro aspecto a notar del diagrama de flujo de la Figura 24 es el comando “S” (STOP), el cual indica a la ESP32-CAM que la última tecla ha dejado de ser presionada y, en consecuencia, debe detener ambos servomotores. Para optimizar el uso de la red, este comando

se envía una única vez, de ahí surge la validación de “si el último comando enviado no es STOP”.

Por otro lado, dentro de la variable “client” se encuentra una instancia del cliente mqtt que genera una conexión al servidor mediante la función “connect” al ser invocada pasando como parámetro la IP del servidor MQTT, el puerto donde corre Mosquitto y un tiempo en segundos de “keepalive” para la conexión (si se supera este tiempo sin conexiones con el bróker, se enviará un ping para mantener la conexión activa). Luego, en un loop infinito se lee una tecla presionada por el usuario y, en caso de que sea alguna de las flechas del teclado, las publica en un tópico “test”, si es la tecla “q”, finaliza el programa. En [este link](#) hay un video donde se muestra su funcionamiento.

### 3.4.2. Suscriptor MQTT desde la ESP32-CAM que controla el auto

En la arquitectura del sistema planteado, la ESP32CAM que controla el movimiento del auto se dispone como “suscriptora” en el tópico “auto” bajo el protocolo MQTT, donde escuchará a la espera de una indicación del usuario para mover el vehículo. Para su implementación se utilizó la librería “<PubSubClient.h>” diseñada para microcontroladores. La misma permite establecer una comunicación MQTT utilizando las siguientes funciones

- setServer(): establece dirección IP del server MQTT y puerto donde corre el servicio.
- setCallback(): establece la función que se ejecutará cuando llegue.
- connect(): hace efectiva la conexión, es decir, envía el comando “Connect Command”.
- subscribe(): suscribe al cliente a un tópico dentro del broke, para lo que envía el comando “Subscribe Request”

En el Apéndice C, “Código del suscriptor desde la ESP32CAM que controla el auto” está su implementación en código.

Si analizamos el tráfico de red con una herramienta como Wireshark se puede ver en la Figura 25 cómo se establece la conexión y se suscribe a un tópico mediante el envío de dichos mensajes. La IP 192.168.129.108 corresponde al ESP32-CAM y la 192.168.129.132 al servidor MQTT

200 12.648996	192.168.129.108	192.168.129.132	MQTT	79 Connect Command
202 12.649286	192.168.129.132	192.168.129.108	MQTT	60 Connect Ack
203 12.654411	192.168.129.108	192.168.129.132	MQTT	65 Subscribe Request (id=2) [test]
204 12.654752	192.168.129.132	192.168.129.108	MQTT	60 Subscribe Ack (id=2)

**Figura 25** – Captura de Wireshark de mensajes para el establecimiento de una conexión MQTT y la suscripción a un tópico

Luego de la conexión, la ESP32-CAM podrá suscribirse a los tópicos que necesite, para esto se utiliza el comando “Subscribe Request [topic]”, también se puede ver esto en los paquetes 203 y 204 de la Figura 25. Es importante notar que ambos cliente (ESP32-CAM) y servidor (Debian VM) se encuentra en una misma red LAN.

Una vez establecida la conexión, si se publican mensajes en el tópico se enviarán al ESP32-CAM desde el servidor, en la Figura 26 se ve cómo ven dichos paquetes viajan bajo el nombre de “publish message”.

305 23.053337	192.168.129.132	192.168.129.108	MQTT	66 Publish Message [test]
315 23.816918	192.168.129.132	192.168.129.108	MQTT	66 Publish Message [test]
319 24.273756	192.168.129.132	192.168.129.108	MQTT	64 Publish Message [test]
323 24.633249	192.168.129.132	192.168.129.108	MQTT	64 Publish Message [test]
332 24.881662	192.168.129.132	192.168.129.108	MQTT	64 Publish Message [test]
345 25.143000	192.168.129.132	192.168.129.108	MQTT	66 Publish Message [test]
356 25.381855	192.168.129.132	192.168.129.108	MQTT	66 Publish Message [test]
359 25.632933	192.168.129.132	192.168.129.108	MQTT	64 Publish Message [test]

**Figura 26** - Captura de Wireshark mensajes del servidor MQTT al suscriptor

En [este link](#) hay un video donde se puede ver la demostración de lo comentado.

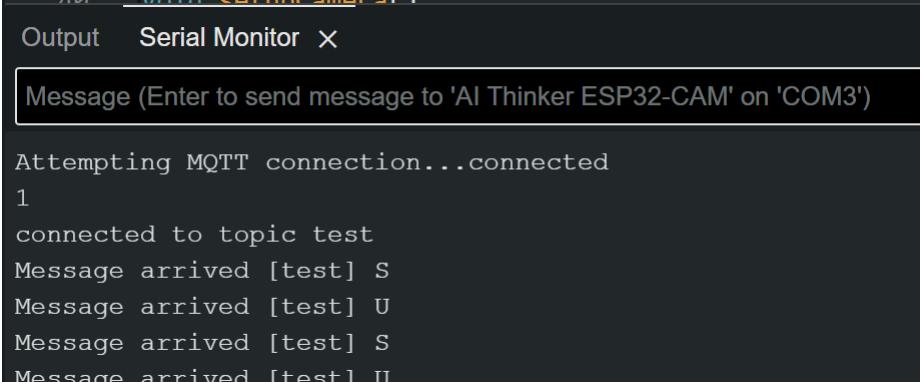
Por otro lado, la directiva client.connected(), tal como su nombre indica, verifica que el cliente este conectado, en el caso de no estarlo, invoca a una función “reconnect()” que dispara nuevamente el client connect de MQTT.

La última directiva que vale la pena mencionar es “client.loop()”, la cual debe incluirse en el loop() principal pues es necesario ejecutarla periódicamente para no perder la conexión al servidor Mosquitto. So inspeccionamos el tráfico utilizando Wireshark vemos que aparecen dos nuevos tipos de mensajes: “Ping Request” por parte del cliente y “Ping Response” por parte del servidor (Figura 27). Su finalidad es la de mantener activa la conexión MQTT por más de que no haya mensajes durante un determinado tiempo.

7889 427.738850	192.168.129.108	192.168.129.132	MQTT	79 Connect Command
7891 427.740278	192.168.129.132	192.168.129.108	MQTT	60 Connect Ack
7893 427.814628	192.168.129.108	192.168.129.132	MQTT	65 Subscribe Request (id=2) [test]
7894 427.816088	192.168.129.132	192.168.129.108	MQTT	60 Subscribe Ack (id=2)
8096 438.035363	192.168.129.132	192.168.129.108	MQTT	66 Publish Message [test]
8129 438.290189	192.168.129.132	192.168.129.108	MQTT	67 Publish Message [test]
8135 438.510065	192.168.129.132	192.168.129.108	MQTT	79 Publish Message [test], Publish Message [test]
8166 442.972260	192.168.129.108	192.168.129.132	MQTT	56 Ping Request
8167 442.973288	192.168.129.132	192.168.129.108	MQTT	60 Ping Response

**Figura 27** – Captura de Wireshark de mensajes para el establecimiento de una conexión MQTT, la suscripción a un tópico, y la mantención de la conexión mediante mensajes Ping Request y Ping Response.

Finalmente, al inspeccionar el monitor serial de Arduino IDE, vemos cómo los mensajes que se envían desde el servidor central llegan efectivamente a la ESP32-CAM (Figura 28).



The screenshot shows the Arduino Serial Monitor window. At the top, there are tabs for "Output" and "Serial Monitor" (which is selected). Below the tabs, a message box contains the text: "Message (Enter to send message to 'AI Thinker ESP32-CAM' on 'COM3')". The main text area displays the following log entries:

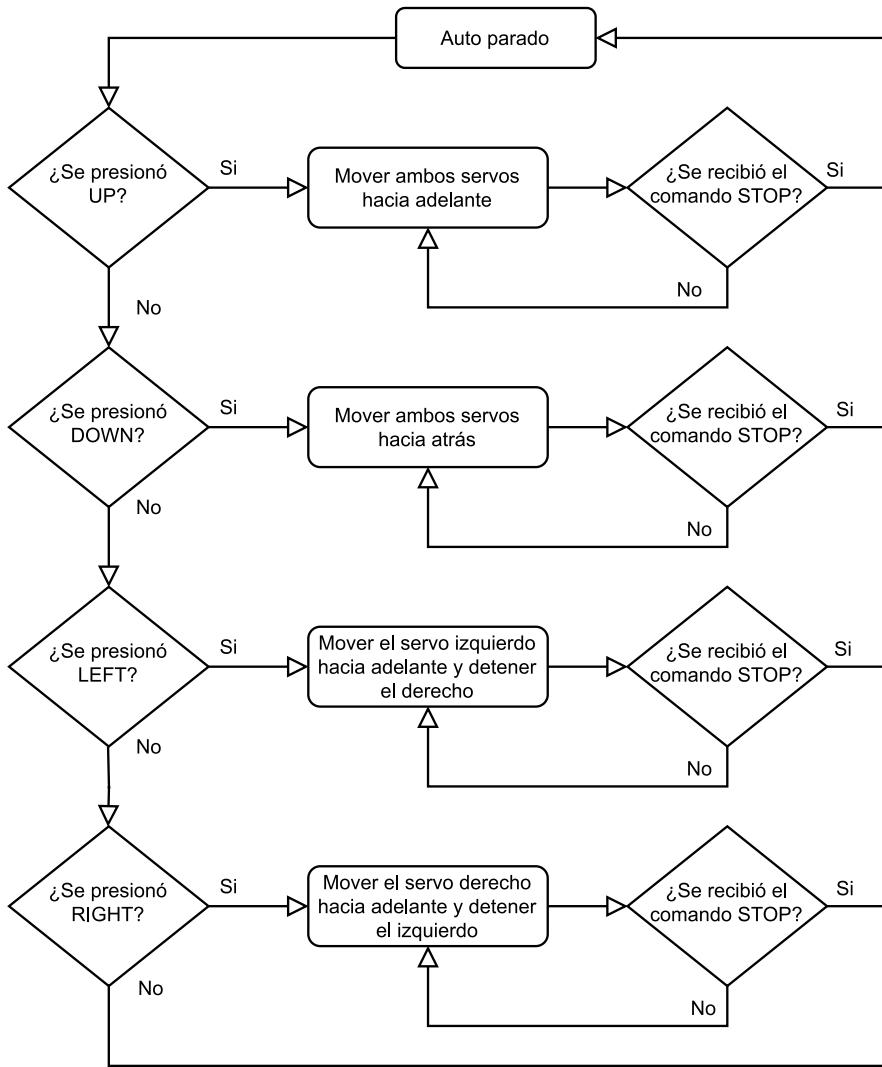
```
Attempting MQTT connection...connected
1
connected to topic test
Message arrived [test] S
Message arrived [test] U
Message arrived [test] S
Message arrived [test] U
```

**Figura 28** – Mensajes en el tópico “test” que llegan a la ESP32-CAM

### 3.5. Control del vehículo

Una vez solucionado el problema de la comunicación entre la ESP32-CAM y el servidor central, el siguiente paso es diseñar el algoritmo de control del vehículo. Para esto es importante saber que, al utilizar la librería <PubSubClient.h> para la comunicación mediante MQTT, se define una función “recall” ejecutada cada vez que llega un mensaje a alguno de los tópicos a los que está suscripta la ESP32-CAM.

Dentro de la función “recall” se diseñó el algoritmo de control del auto. La lógica del mismo es: si se presiona UP, ambos servos rotarán hacia adelante, si se presiona DOWN ambos servos rotaran hacia atrás, si se presiona LEFT rotará únicamente el servo izquierdo, y si se presiona RIGHT rotará únicamente el servo derecho. En el diagrama de flujo de la Figura 29 se puede ver gráficamente cómo es el funcionamiento de este algoritmo

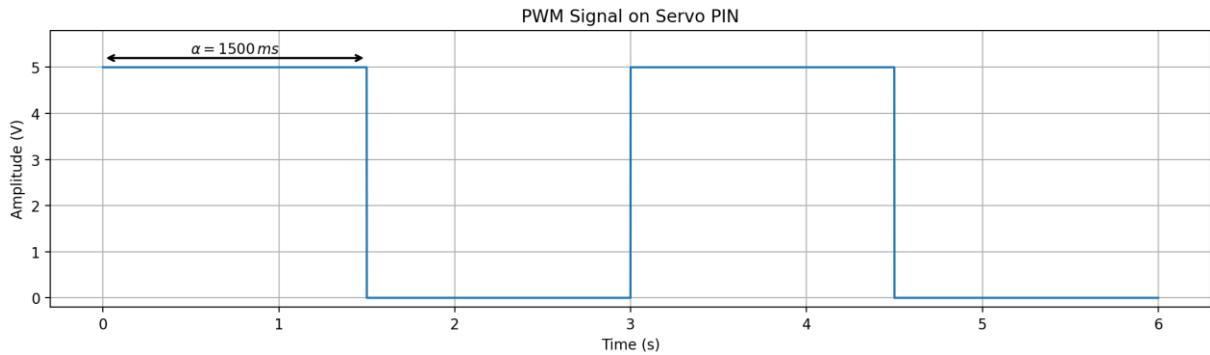


**Figura 29** – Diagrama de flujo del publicador a la ESP32-CAM que controla el auto

Al igual que sucedía en el publicador, en el suscriptor se incluye la lógica necesaria para detener el movimiento del auto como se puede ver en los bloques de decisión “¿Se recibió el comando STOP?”. Recordemos que dicho comando se enviará por parte del servidor central una vez que el usuario deje de presionar alguna de las teclas.

En el Apéndice B, “Código de la función recall para el control del auto”, se puede ver el código donde se implementa dicho algoritmo. Aquí, lo que en el diagrama de flujo era “Mover en X dirección” se logra con la función `servoX.writeMicroseconds( $\alpha$ [ms])`, donde `servoX` es una instancia de un servomotor definida en la librería `<ESP32Servo.h>`. Al llamar a la función `writeMicroseconds()`, se escribe en la salida del PWM de `servoX` un pulso de ancho  $\alpha$  ms. En el caso de los servos que rotan 360 grados, al usar esta función esto provoca que en la salida del pin asociado al servo, se genere un pulso PWM periódico con ancho  $\alpha$  ms, provocando que de este modo el servo gire con una velocidad angular constante. En la Figura

30 se ve representación gráfica hecha con matplotlib donde se muestra la onda cuadrada ideal en la salida del pin del servo.



**Figura 30** – Representación usando matplotlib de un pulso PWM con un ancho  $\alpha = 1500ms$

Según el ancho de  $\alpha$  el servo rota hacia adelante, hacia atrás o se queda pareado. Idealmente los valores responden al siguiente comportamiento:

- Pulso de 1500ms: El servo no rota.
- Pulso > 1500ms: El servo rota en una dirección.
- Pulso < 1500ms: El servo rota en dirección opuesta al pulso > 1500ms.

Se dice que “idealmente” puesto que para los servos provisionados por la cátedra el valor de referencia de 1500ms está más cercano a los 1480ms. A medida que avanza el desarrollo cuestiones de este tipo de “fine tuning” se irán optimizando.

Finalmente, en el [este link](#) se puede ver al auto en funcionamiento con todas las consideraciones previamente mencionadas.

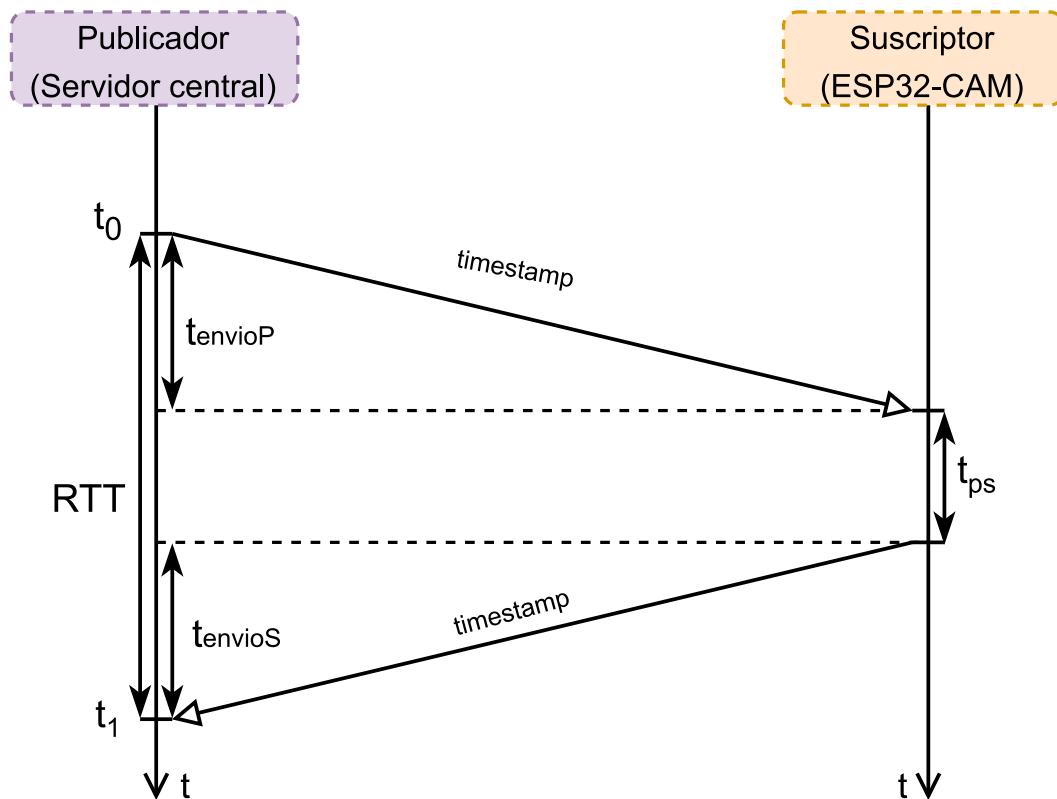
### 3.6. Medición de tiempos del protocolo MQTT para ESP32-CAM que controla el movimiento del auto

La medición de tiempos en los protocolos como MQTT es crucial en proyectos que involucran el procesamiento del stack TCP/IP en microcontroladores. Esto se debe a que cualquier retraso o pérdida de datos puede resultar en decisiones incorrectas o tardías, lo que tiene podría tener un alto impacto en el desempeño total del sistema. Por lo tanto, la capacidad para medir y optimizar los tiempos del protocolo de capa de aplicación utilizado es indispensable para garantizar la integridad y la rapidez en aquellos sistemas donde intercambio de mensajes es la base de funcionamiento.

En el caso de MQTT y la ESP32-CAM que controla el movimiento del auto, se realizaron dos mediciones del RTT promedio de 100 mensajes desde el servidor: una con la ESP32-CAM con carga y otra sin. La que tiene carga procesa el movimiento del servo y publica

sus imágenes en un servidor web, mediante que la que no tiene mide únicamente la latencia de la red y el protocolo.

En la Figura 31 se puede ver el diagrama de tiempos de cómo se da la comunicación entre el servidor central y el suscriptor. El mecanismo para medir el round trip time (RTT) de la red se basa en enviar un mensaje a la ESP32-CAM con el tiempo actual del servidor,  $t_0$ . Luego, la ESP32-CAM recibe dicho mensaje y lo reenvía al servidor luego de un tiempo de procesamiento  $t_{ps}$ , este tiempo dependerá de la carga del microcontrolador. Finalmente, el mensaje con contenido  $t_0$  llega al servidor en un tiempo  $t_1$ .



**Figura 31** – Diagrama de tiempo de la comunicación entre el publicador y el suscriptor

Del diagrama de la Figura 31 nos interesa el cálculo de tres valores que nos darán una métrica del rendimiento de nuestro sistema:  $RTT$ ,  $t_{envioP}$ ,  $t_{envios}$ . El primero deriva fácilmente de la medición de  $t_1$  y  $t_2$ .

$$RTT = t_1 - t_0$$

A su vez, sabemos que

$$RTT = t_{envioP} + t_{envios} + t_{ps}$$

Para poder resolver este cálculo haremos dos asunciones:  $t_{envioP} = t_{envios} = t_{envio}$ , lo cual es coherente porque ambos mensajes viajan bajo el mismo medio de transmisión, y que, inicialmente,  $t_{ps-} = 0$ , pues se evaluará el sistema sin carga. De este modo, se tiene

$$t_{envio} = \frac{RTT}{2} = \frac{t_1 - t_0}{2}$$

Luego de calcular el RTT promedio de 100 mensajes el resultado fue

$$RTT = 16.66 \text{ ms}$$

De donde se concluye que

$$t_{envio} = 8.33 \text{ ms}$$

La siguiente medición que se realizó fue la del RTT promedio con el ESP32-CAM con carga, es decir,  $t_{ps} > 0$  y se midió que

$$RTT = 33.89 \text{ ms}$$

Por lo que se concluye que

$$t_{ps} = RTT - 2 * t_{envio} = 17.23 \text{ ms}$$

En ambos casos de prueba el tiempo de respuesta no se vio críticamente comprometido para lo que es la funcionalidad del sistema, por lo que, por más que el tiempo de procesamiento en la ESP32-CAM aumenta significativamente conforme lo hace la carga, el sistema sigue cumpliendo los requerimientos.

## 4. Documentación Relacionada

[Enlace a la bitácora](#)

[Repositorio del código principal](#)

[Repositorio de pruebas](#)

[Videos de demostración movimiento de la torreta e interfaz grafica](#)

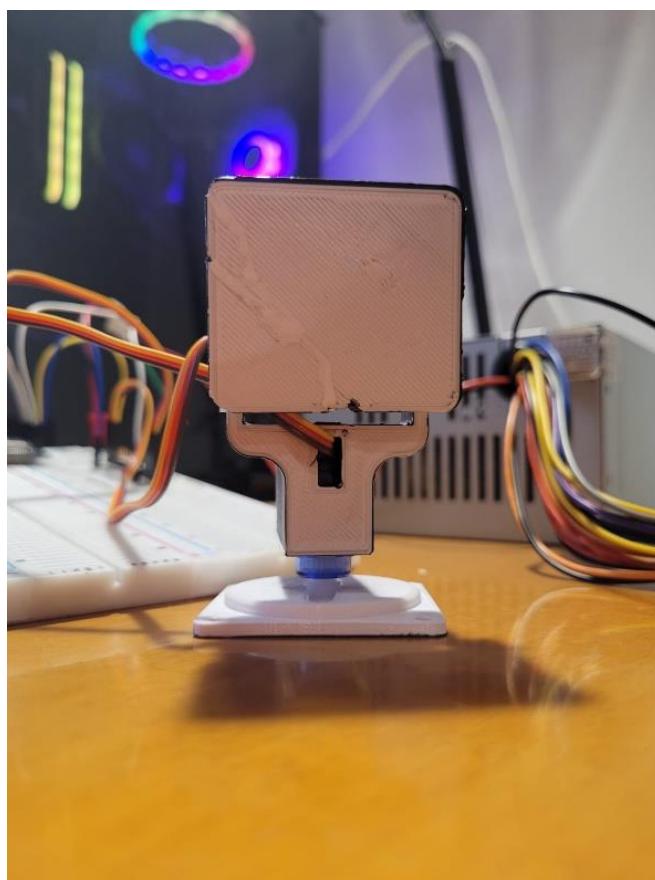
[Video de demostración del movimiento del auto](#)

[Video de demostración del funcionamiento del Web Server](#)

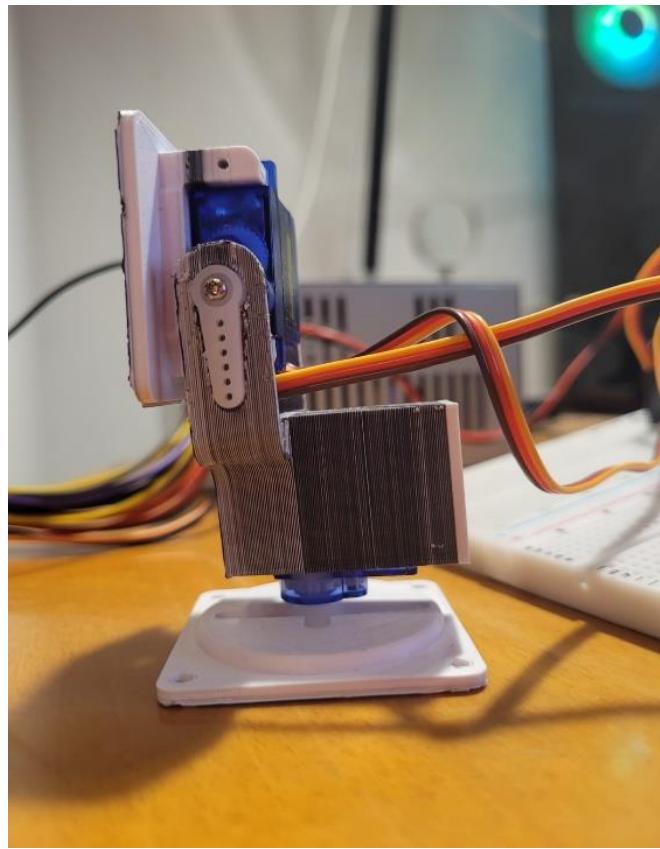
[Enlace video demostración de servo y giroscopio](#)

[Enlace al código completo del servo y el giroscopio](#)

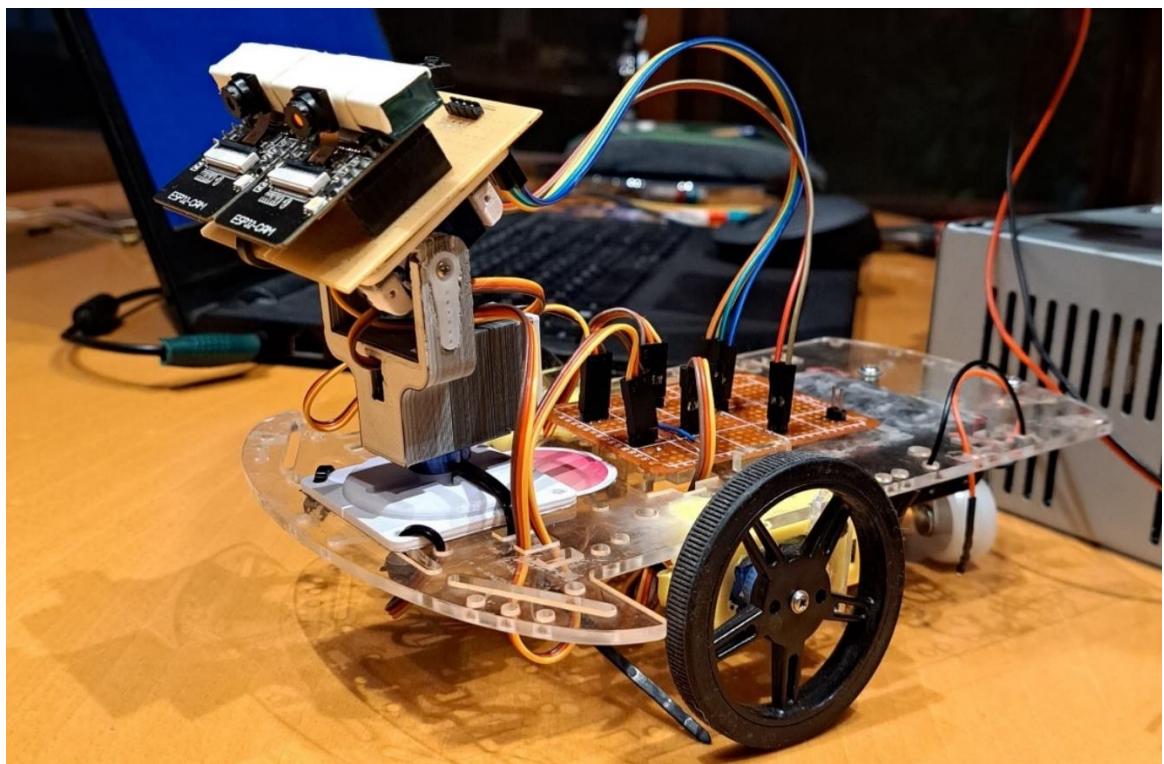
[Muestra final del proyecto](#)



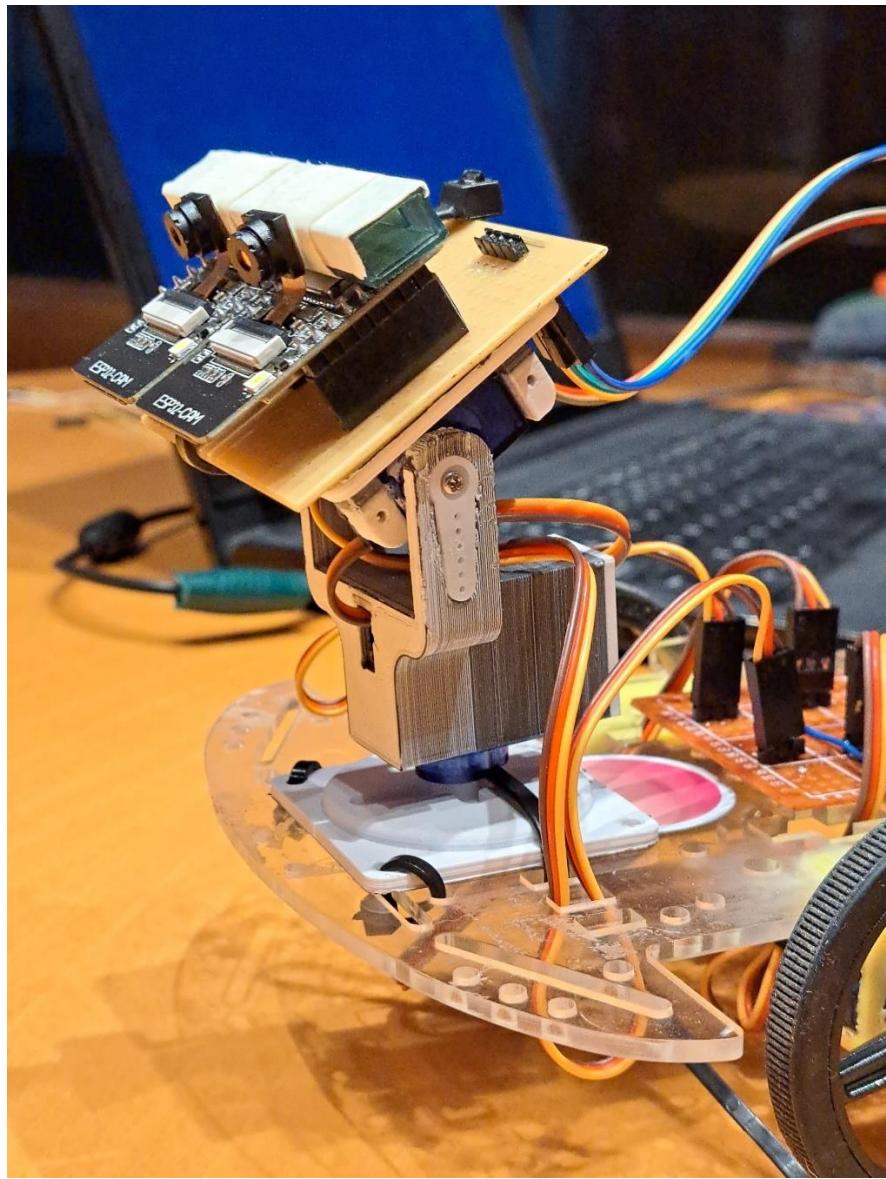
**Figura 32 – Torreta I**



**Figura 33** – Torreta II



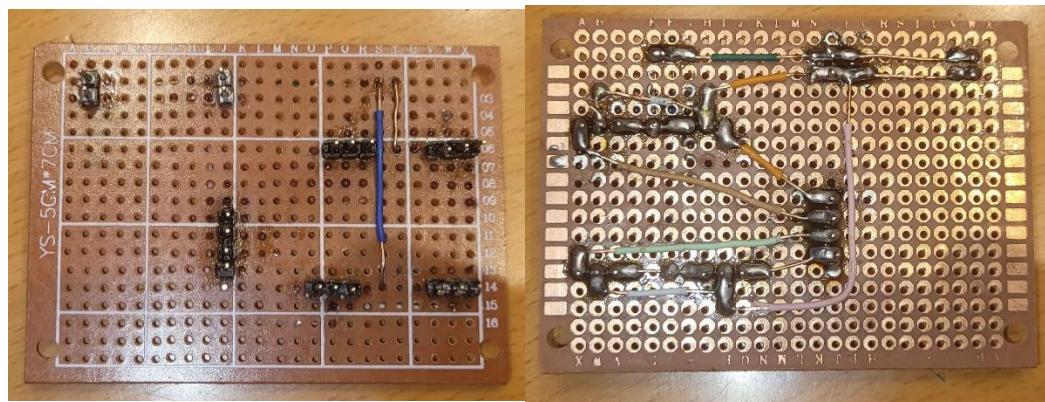
**Figura 34** – Vehículo final montado



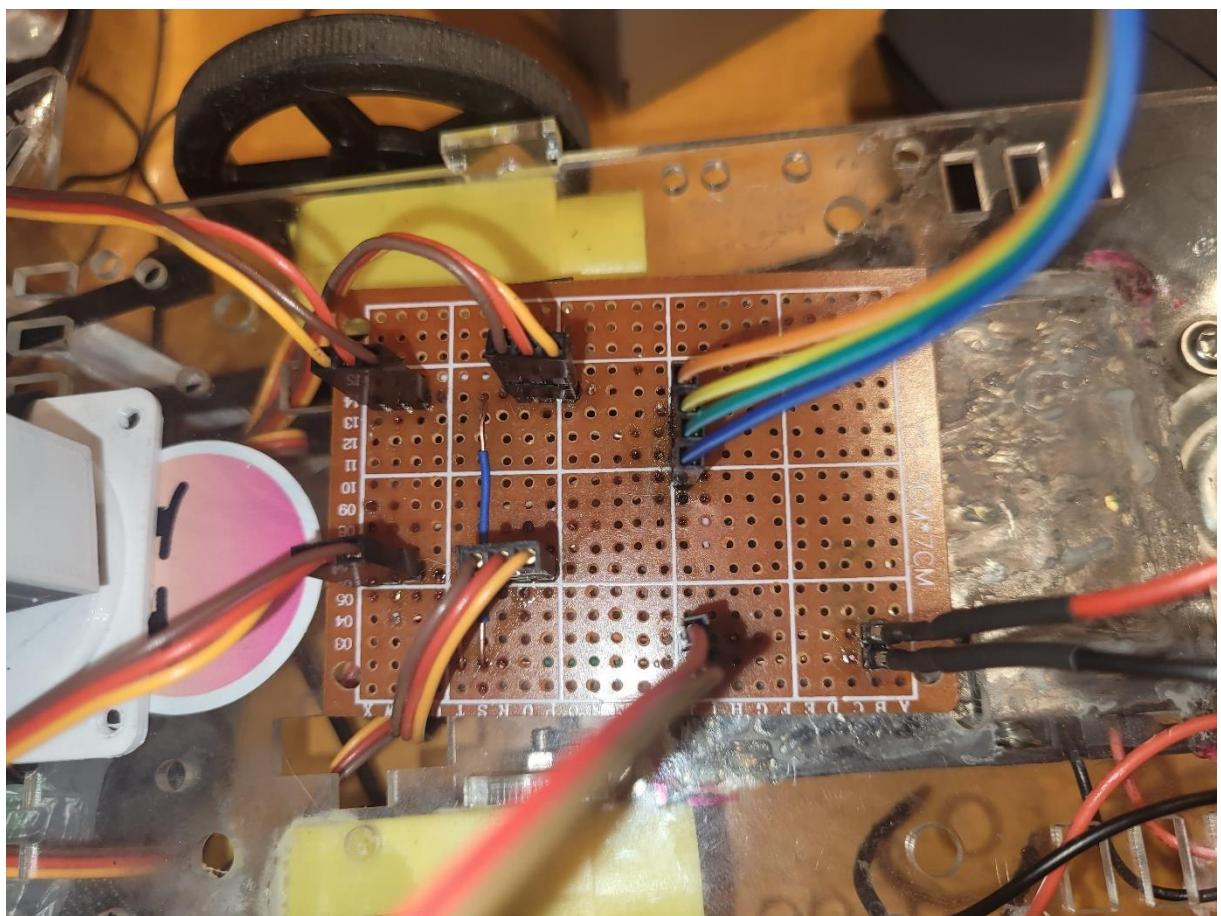
**Figura 35** – ESP32-CAMs montadas sobre la torreta

#### 4.1.1. Placas desarrolladas

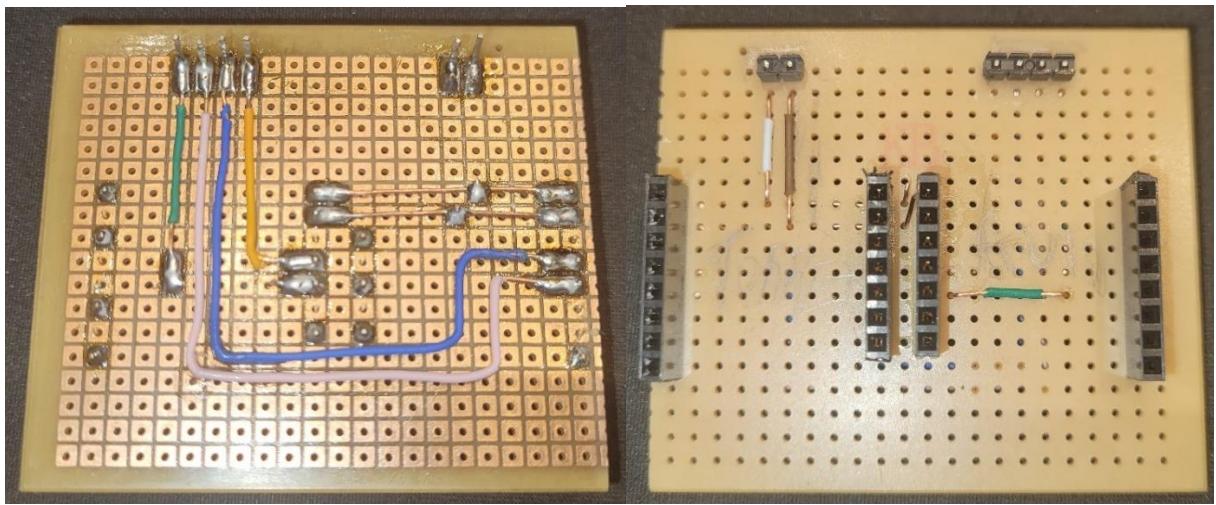
Tras el diseño de las PCB, se llevo a cabo la fabricación de las mismas sobre placas perforadas. En ellas, los agujeros están pre-perforados por lo que los componentes se conectan mediante el soldado de puentes. A pesar de que estas placas no se corresponden con el término "Printed Circuit Board" (PCB) en el sentido estricto, ya que no presentan circuitos impresos, cumplen con las funcionalidades necesarias y satisfacen todos los requerimientos técnicos especificados para el proyecto, por lo que son completamente adecuadas para su propósito. A continuación, se presentan las imágenes de las placas finalizadas.



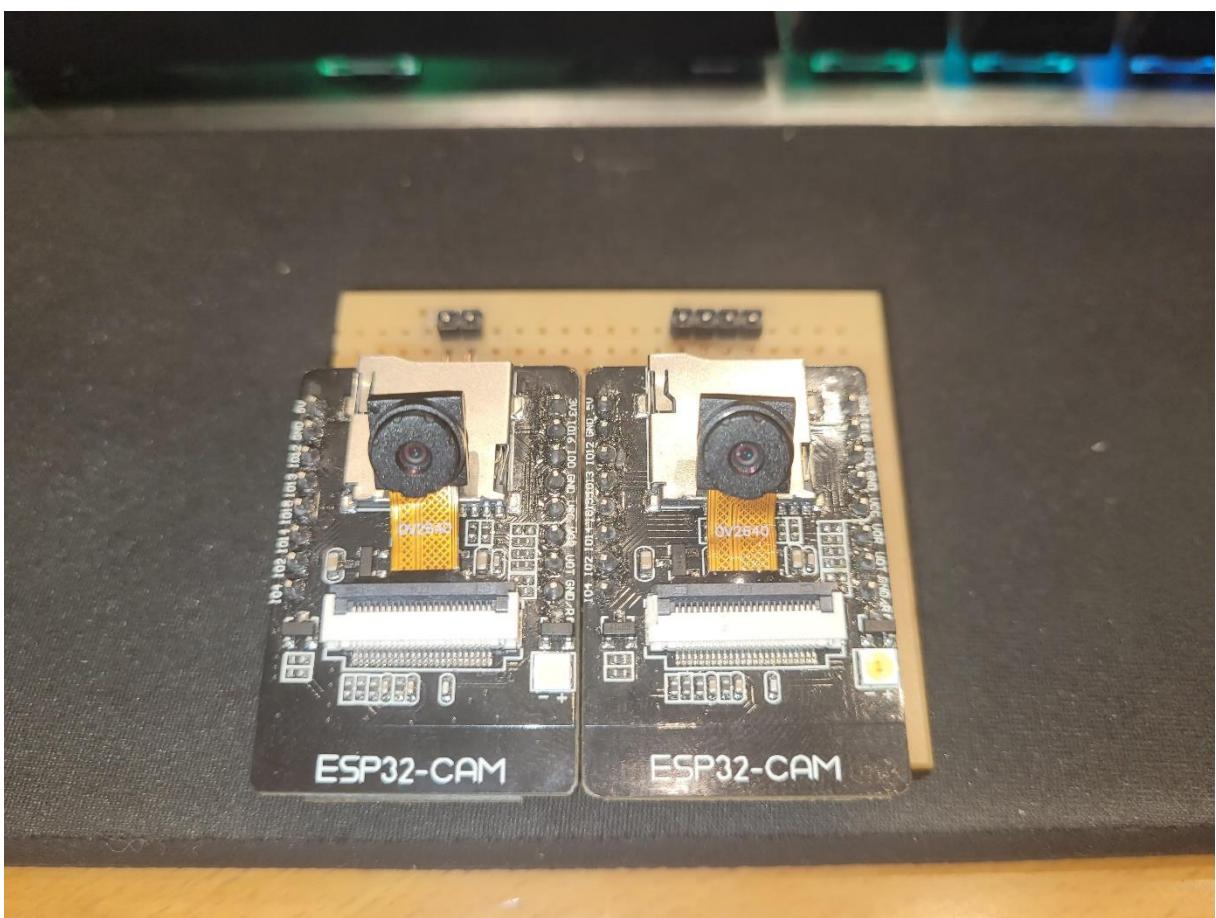
**Figura 36** – “Placa Auto” construcción física



**Figura 37** – “Placa Auto” montada en el vehículo



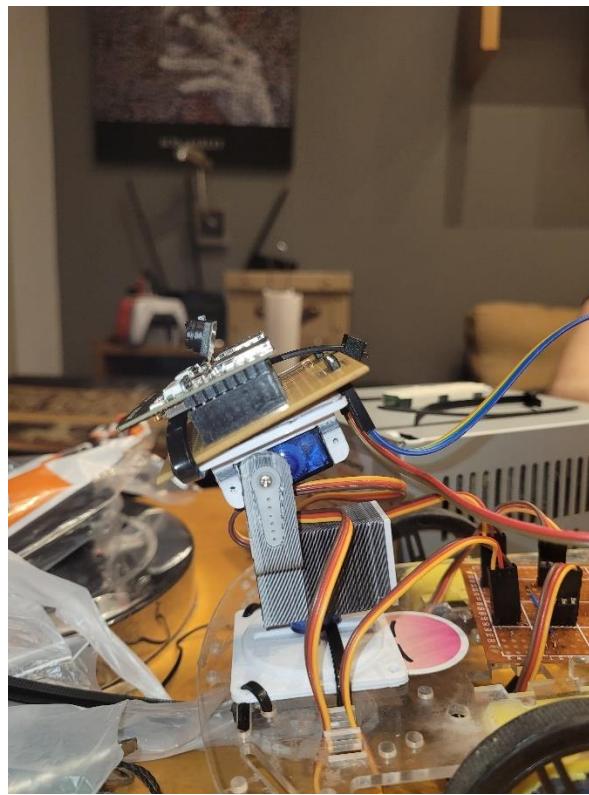
**Figura 38** – “Placa Torreta” construcción física



**Figura 39** – “Placa Torreta” con los ESP32-CAM montados



**Figura 40** – “Placa Torreta” montada sobre la torreta



**Figura 41** – “Placa Torreta” montada sobre la torreta

## Apéndice A: Materiales y Presupuesto

Debe aclararse que todos los materiales necesarios para la realización del proyecto fueron ya provistos por la cátedra, de todos modos, en la Tabla A1 se puede ver la lista de los mismos.

Tabla A1 – Presupuesto

Hardware	Especificaciones	Cantidad	Valor Unitario	Valor Total	Link de compra
ESP32 Cam	-	2	\$ 9.059	\$ 18.118	<a href="#">Mercado Libre (ML)</a>
Programador para ESP32 Cam	-	1	\$ 3.532	\$ 7.064	<a href="#">ML</a>
Servomotores	De entre 1.5 kg/cm a 12 kg/cm	2	\$ 2.553	\$ 5.106	<a href="#">ML</a>
Gafas VR	-	1	\$ 10.448	\$ 10.448	<a href="#">ML</a>
Giroscopio	MPU 6050	1	\$ 2.580	\$ 2.580	<a href="#">ML</a>
NodeMCU Esp8266	-	1	\$ 4.380	\$ 4.380	<a href="#">ML</a>
Esqueleto del vehículo + Motores + Ruedas	-	1	-	-	-

Presupuesto total (sin contar esqueleto del vehículo + ruedas + motores): \$47696

## Apéndice B: Otros protocolos y desarrollos futuros

### B.1 Protocolo RTSP sobre UDP

Una de las alternativas que se analizó para transmitir video fue el protocolo de capa de aplicación Real Time Streaming Protocol (RTSP) dado que el mismo trabaja sobre UDP y, dado que este protocolo de capa de transporte no es orientado a conexión y es mucho más ligero en contraposición de TCP, parecía una buena alternativa a transmitir vía HTTP/TCP. De este modo, basándonos en [este repositorio](#) que utiliza la librería Micro-RTSP, se logró levantar un servidor RTSP en una de las ESP32CAMs.

Primero cargamos el código en la ESP32CAM y verificamos la IP del servidor mediante el monitor serial de Arduino IDE (Figura X)

```
Output Serial Monitor X

Internal Total heap 243604, internal Free Heap 218612
SPIRam Total heap 2095087, SPIRAM Free Heap 2094827
ChipRevision 3, Cpu Freq 240, SDK Version v4.4.5
Flash Size 4194304, Flash Speed 80000000
#####
..
WiFi connected with IP 192.168.137.76
Stream Link: rtsp://192.168.137.76:8554/mjpeg/1

RTSP task up and running
```

**Figura B1** – Monitor serial de Arduino IDE que muestra la IP del servidor RTSP

Una vez confirmamos que el código se cargó exitosamente, accedemos con un cliente RTSP, como puede ser VLC y vemos cómo en la Figura X se imprimen los comandos recibidos a través de RTSP en la consola serial.

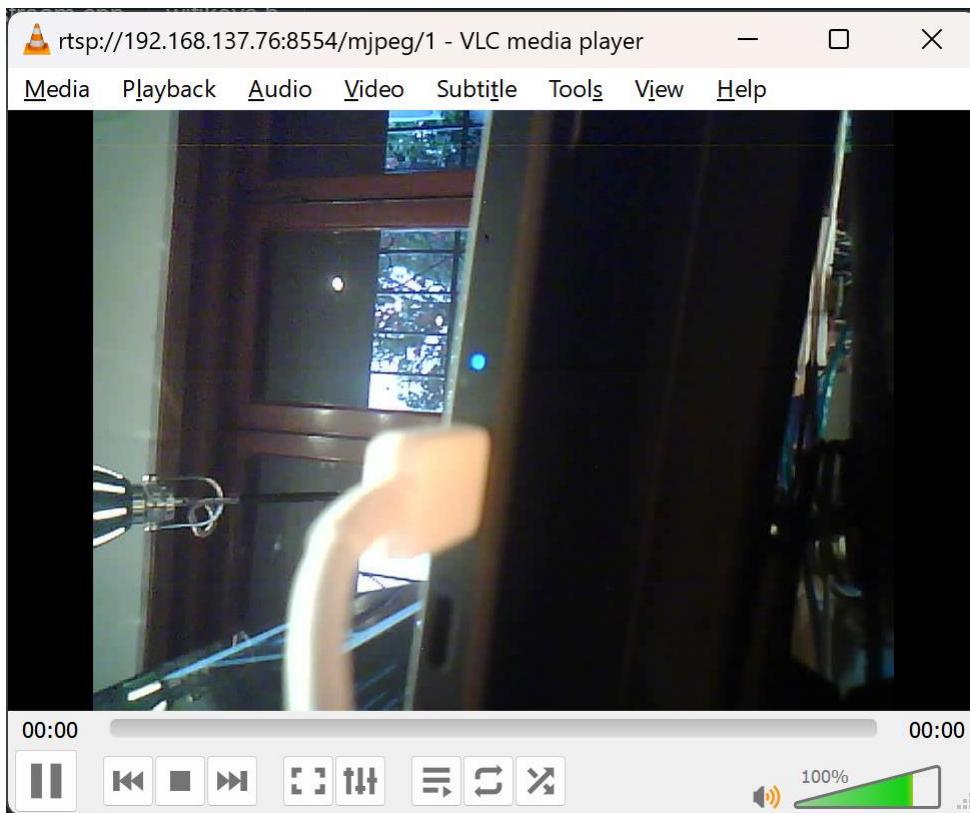
```
Output Serial Monitor X

WiFi connected with IP 192.168.137.76
Stream Link: rtsp://192.168.137.76:8554/mjpeg/1

RTSP task up and running
RTSP client started connection
Creating TSP streamer
Created streamer width=800, height=600
Creating RTSP session
RTSP received OPTIONS
RTSP received DESCRIBE
RTSP received SETUP
RTSP received PLAY
```

**Figura B2** – Monitor serial de Arduino IDE que muestra los comandos recibidos mediante RTSP

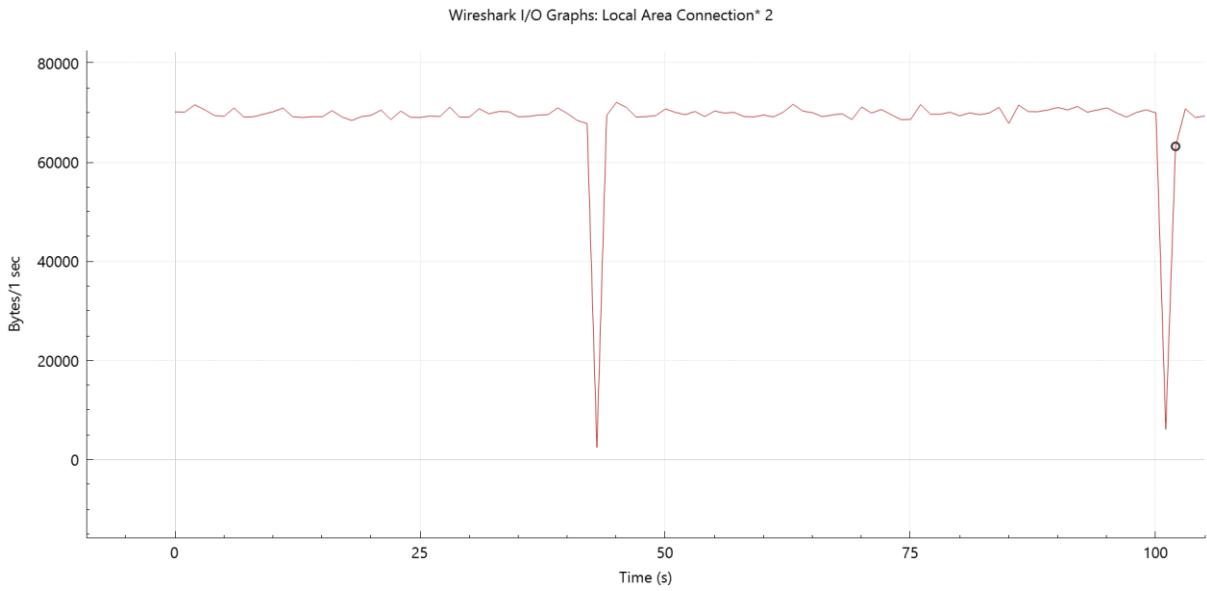
Al acceder con mediante el cliente RTSP se puede ver el stream de video como en la Figura X.



**Figura B3** – Cliente de VLC para RTSP accediendo al streaming de video

Al comparar el rendimiento en términos de fotogramas por segundo que se obtienen utilizando RTSP es significativa la diferencia con HTTP, a tal punto que el primero afecta significativamente el funcionamiento del sistema planteado. Para observar lo mencionado se sugiere al lector ver los siguientes videos: [Funcionamiento de RTSP](#) vs [Funcionamiento de HTTP](#).

Si analizamos mediante Wireshark la cantidad de bytes que se mandan por segundo encontramos que estos son 70 kB por segundo tal como se puede ver en la Figura X, mientras que en TCP teníamos 410 kB por segundo, lo que produce que la fluidez en RTSP sea mucho menor en comparación a TCP.



**Figura B4** – Cantidad de Bytes por segundo desde la ESP32-CAM mediante RTSP a lo largo del tiempo

Se concluye entonces que, dado que teóricamente los protocolos teóricamente deberían funcionar mejor, hay un problema en la implementación práctica, particularmente en la librería utilizada “Micro-RTSP”. Resulta que es una de las muy pocas que hay desarrolladas para este protocolo para las ESP32-CAMs y no dispone de una comunidad útil. Se plantea entonces como posible trabajo futuro de continuación del trabajo la implementación de este protocolo para mejorar aún más la latencia en la transmisión de video, proyectando el desarrollo de más librerías para RTSP para las ESP32-CAMS.

## B.2 Servidor NTP para la sincronización de los dispositivos de la red

En este proyecto se utilizan únicamente cuatro dispositivos conectados la red y, como la misma es local y no presenta grandes latencias, la sincronización entre los dispositivos no es un problema. De todos modos, se plantea una alternativa que permite al sistema tener una escalabilidad sostenible para el caso en que se agreguen más dispositivos y/o se encuentre en circunstancias donde la latencia sea significativamente mayor. En tales casos, sería necesaria la sincronización de los relojes locales de los dispositivos y la comunicación de estos relojes como información adicional, para así saber cuán “atrasado” está un paquete. Para que todos los dispositivos tengan el mismo horario, es necesario que todos consulten al mismo servidor, para ello se utiliza el protocolo Network Time Protocol (NTP)

NTP es un protocolo de red utilizado para sincronizar los relojes de los sistemas informáticos a través de paquetes de red. Su función principal es asegurar que todos los dispositivos en una red estén sincronizados en tiempo, lo cual es crucial para la ejecución de operaciones precisas y coordinadas en la red.

Un servidor NTP conocido y ampliamente utilizado es “Chrony” por su ligereza y eficiencia. En este proyecto, se desplegó un servidor Chrony en el servidor central y se testeó su funcionamiento. En la Figura X se ve el servicio en funcionamiento en la máquina virtual.

```

Terminal - admin-autito@proyecto-autito: ~
File Edit View Terminal Tabs Help
admin-autito@proyecto-autito:~$ sudo systemctl status chrony
● chrony.service - chrony, an NTP client/server
  Loaded: loaded (/lib/systemd/system/chrony.service; enabled; preset: enabled)
  Active: active (running) since Mon 2023-12-04 18:59:49 EST; 3s ago
    Docs: man:chronyd(8)
          man:chronyc(1)
          man:chrony.conf(5)
   Process: 742 ExecStart=/usr/sbin/chronyd $DAEMON_OPTS (code=exited, status=0/SUCCESS)
 Main PID: 744 (chronyd)
    Tasks: 2 (limit: 4713)
   Memory: 972.0K
      CPU: 96ms
     CGroup: /system.slice/chrony.service
             └─744 /usr/sbin/chronyd -F 1
               ├─745 /usr/sbin/chronyd -F 1

Dec 04 18:59:49 proyecto-autito systemd[1]: Starting chrony.service - chrony, an NTP client/server.>
Dec 04 18:59:49 proyecto-autito chronyd[744]: chronyd version 4.3 starting (+CMDMON +NTP +REFCLOCK >
Dec 04 18:59:49 proyecto-autito chronyd[744]: Frequency 7.685 +/- 0.101 ppm read from /var/lib/chrono>
Dec 04 18:59:49 proyecto-autito chronyd[744]: Using right/UTC timezone to obtain leap second data
Dec 04 18:59:49 proyecto-autito chronyd[744]: Loaded seccomp filter (level 1)
Dec 04 18:59:49 proyecto-autito systemd[1]: Started chrony.service - chrony, an NTP client/server.
lines 1-21/21 (END)

```

**Figura B5** – Cantidad de Bytes por segundo desde la ESP32-CAM mediante RTSP a lo largo del tiempo

Utilizando la librería en Python “ntplib” se logró establecer conexión con el servidor y obtener el horario con resolución de milisegundos, tal como se puede ver en la Figura X.

```

>>> from datetime import datetime
>>> import ntplib
>>> NTP_SERVER = '192.168.137.48'
>>> client = ntplib.NTPClient()
>>> response = client.request(NTP_SERVER, version=3)
>>> print(datetime.fromtimestamp(response.tx_time))
2023-12-04 21:19:05.981341

```

**Figura 30** – Conexión al servidor NTP

Una vez hecho esto, fue necesario agregar un “timestamp” bajo el nombre de “X-NTPTimestamp” en los paquetes TCP enviados por la ESP32-CAM, de modo que el servidor central pueda comprarar la diferencia entre ambos tiempos, como la ESP32-CAM obtendrá el video del servidor NTP no habría problemas respecto a sincronización de relojes. Para hacer las peticiones desde la ESP32-CAM se utilizó la librería NTPClient.h.

En el código que se adjunta junto con este informe, en el archivo “mediciones/mediciones\_NTP” hay una implementación en la cual se hacen peticiones al

stream de video de una de las dos ESP32-CAM, se extrae el header “X-NTPTimestamp” y se intenta realizar la resta entre ambos valores. Aquí surgió un problema por lo cual este desarrollo no se incluyó en la funcionalidad final del proyecto, resulta que la librería desarrollada para la ESP32-CAM de NTP no permite resolución de milisegundos, por lo que el resultado de la resta mencionada no se correspondía con el valor real de la diferencia de tiempos. Tras investigar otras librerías no se llegó a ninguna alternativa viable, por lo que se deja como trabajo futuro la implementación de una librería que permita la resolución en milisegundos desde las ESP32-CAMs. Esto permitiría sincronizar los dispositivos (aún más) en la red y tener información acerca de la latencia en el momento en que llega cada paquete a su destino.

## Apéndice C: Pruebas y desarrollos

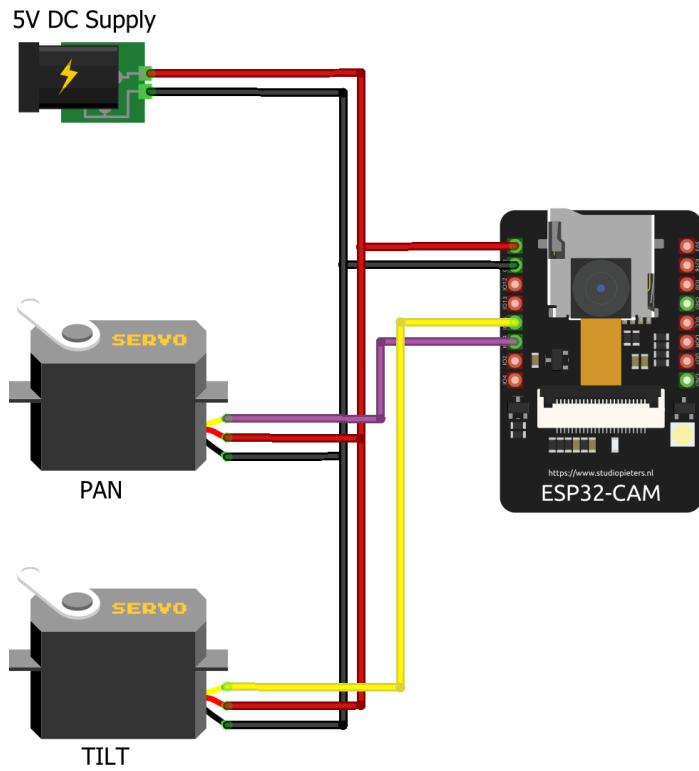
### C.1 Desarrollo y configuración de la Torreta

En las etapas iniciales del desarrollo del proyecto, se prestó una meticulosa atención al diseño y construcción de la torreta, la cual proporciona dos grados de libertad y actúa como una plataforma móvil para las cámaras ESP32-CAM. La estructura de esta torreta fue fabricada mediante impresión 3D, utilizando plástico como material principal debido a su adecuada relación entre peso, resistencia y precio. El archivo de impresión puede encontrarse en el siguiente [link](#)<sup>1</sup>.

Posterior a la impresión, se procedió con la integración de los servomotores, los cuales permiten los movimientos de "pan" y "tilt" (rotación horizontal y vertical, respectivamente) de la torreta. El diagrama de conexión física se encuentra a continuación:

---

<sup>1</sup> Link al archivo 3d de la torreta: <https://www.thingiverse.com/thing:2038433/files>



**Figura C1 5** – Conexionado físico de los servomotores de la torreta y la ESP32 a la fuente. La fuente utilizada es de una PC aportada por la cátedra

Una vez colocados, para poder mover la torreta correctamente, fue necesario determinar ciertos parámetros de los servomotores SG90 como, por ejemplo, el rango de ancho de pulso, el ángulo de rotación y la frecuencia a la que trabaja. Todos estos datos fueron extraídos de la hoja de datos<sup>2</sup>.

A lo largo del proceso de desarrollo, se identificó una limitación física respecto al rango de movimiento vertical de la torreta debido al diseño de la impresión. Esto derivó en la necesidad de implementar ajustes en el software de control para no romper la estructura. Para resolver este inconveniente, se desarrolló un código específico destinado a controlar el ángulo de rotación vertical del servomotor. El software, hace rotar el servomotor de 0 hasta una variable definida llamada X. Cambiando el valor de inicialización de esta variable, se puede deducir cual es el máximo rango soportado por la estructura. El código puede observarse a continuación:

Esta implementación reveló que el movimiento vertical de la torreta tendrá un rango operativo de 0 a 130 grados, estableciendo una limitación para los movimientos en este eje. +

---

<sup>2</sup> Link a las datasheets del servomotor: <https://datasheetspdf.com/pdf-file/791970/TowerPro/SG90/1>

```

#include <ESP32Servo.h>

Servo myservo; // Crea un objeto servo para controlar un servomotor

const int servoPin = 13; // El pin al que está conectado el servomotor

int x = 180; // Define el límite de rotación

void setup() {
    myservo.setPeriodHertz(50); // Frecuencia de 50Hz
    myservo.attach(servoPin, 500, 2400); // Rango de ancho de pulso
}

void loop() {
    for (int pos = 0; pos <= x; pos++) { // Va de 0 grados a x grados
        myservo.write(pos);
        delay(500); // Delay de 0.5 segundos
    }
}

```

## C.2 Pruebas preliminares con un servomotor

Una vez lograda una correcta lectura de los datos, procedimos con las primeras pruebas que integraran al MPU6050 y un servo. Si bien, en el proyecto final no se implementa con una conexión física de los servos a la placa utilizada para la lectura del giroscopio, sino que se proyecta que los datos se envíen a través de una conexión inalámbrica, resultaba importante observar si los datos obtenidos mediante el procesamiento previamente enseñado eran de utilidad para los servos.

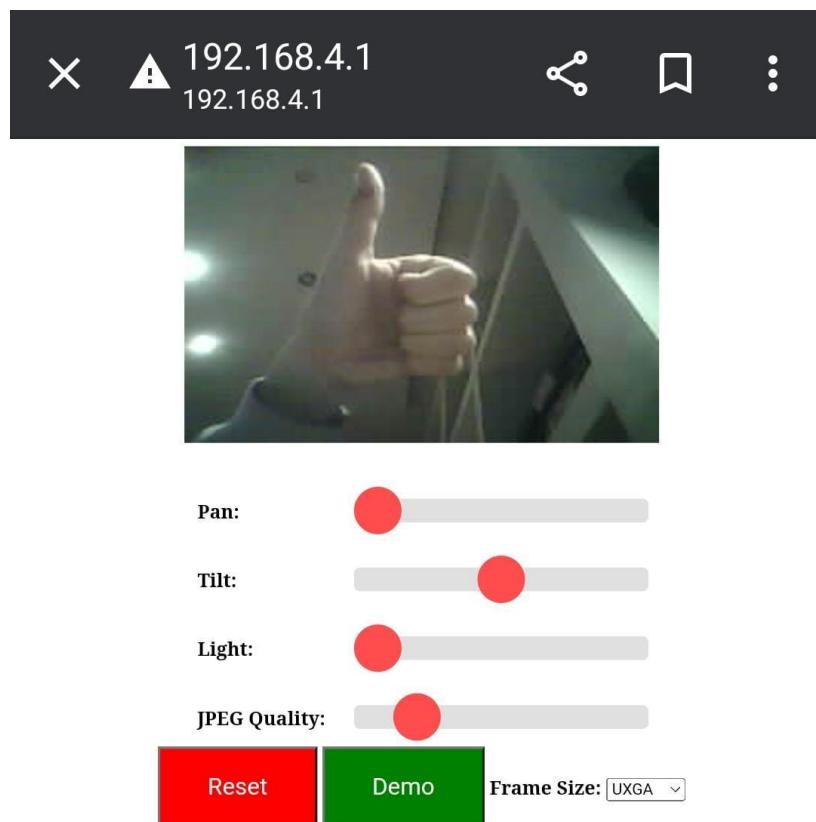
Es así que se estructuró una prueba en la que se tomara la rotación sobre un único eje y ésta se viera plasmada en el movimiento de un único servomotor. En la Figura 7 se puede observar el diseño esquemático de esta prueba, mientras que en la sección Documentación Relacionada se podrán encontrar fotografías del circuito electrónico.

## C.3 Prueba inicial: Desarrollo del WebServer y Streaming de Video con TCP

Para estas primeras pruebas se utilizó solo 1 cámara, se seleccionó el protocolo TCP para la transmisión del streaming y el webserver se hosteo dentro del mismo ESP32 con el objetivo de facilitar la implementación y el diagnóstico en las fases iniciales del desarrollo. Cabe recalcar que esta no será la estructura final del proyecto, el protocolo TCP garantiza la entrega de los paquetes de datos confirmando cada uno de ellos y mantiene el orden de los mismos. Esto hace que no sea un protocolo rápido ni eficiente para el streaming de video. Una de las principales opciones a analizar en el corto plazo es el protocolo UDP que no envia ningún

tipo de confirmación de recepción. Así mismo, en próximas pruebas analizaremos el rendimiento de hostear el webserver en la PC principal, reduciendo la carga del ESP32.

El servidor web, desplegado en el ESP32, no solo maneja la transmisión de video, sino que también proporciona una interfaz de usuario gráfica (GUI) accesible a través de un navegador web estándar. Esta interfaz, construida utilizando HTML5 y JavaScript, no solo muestra el stream de video, sino que también permite al usuario controlar la la rotación de los servomotores de la torreta. A continuación, se observa un screenshot de la misma



**Figura C2 – Interfaz grafica**

La interfaz gráfica de usuario incluye elementos como sliders para controlar los servos de pan, tilt, la intensidad del led integrado (utilizado únicamente como verificación de la comunicación entre la página web y la placa) y la calidad de video. También, contiene un visor para el stream de video, y controles adicionales como botones para resetear la posición de la torreta o hacer una “demo” mostrando los movimientos que es capaz de hacer. Adicionalmente, podemos encontrar un dropdown en el que se puede configurar el frame size de la camara (estas funcionalidades aún se encuentran en desarrollo). El código relacionado a esta GUI se encuentra en el ANEXO 1 con sus debidos comentarios.

Una vez que la cámara captura un fotograma, este se envía a través del WebSocket a la interfaz del usuario en el navegador web. Esta transmisión del frame se realiza en formato binario, lo cual significa que los datos de la imagen se envían tal como están en memoria (sin codificación adicional como Base64), reduciendo la latencia y la carga computacional en el ESP32.

```
//capture a frame
camera_fb_t * fb = esp_camera_fb_get();
if (!fb)
{
    Serial.println("Frame buffer could not be acquired");
    return;
}

//Envia el fotograma en formato binario al websocket de la camara
wsCamera.binary(cameraClientId, fb->buf, fb->len);
// Devuelve un frame buffer al controlador de la cámara después de que se ha terminado de usar,
// permitiendo que el buffer de memoria pueda ser reutilizado para almacenar futuros frames capturados
// por la cámara.
esp_camera_fb_return(fb);
```

Es fundamental mencionar que la gestión eficiente de la transmisión de datos es crucial para mantener una experiencia de usuario suave y libre de interrupciones, especialmente en aplicaciones que utilizan streaming de video. En este contexto, el código implementa un mecanismo de control de flujo para asegurar que los datos se envíen al cliente a una velocidad que pueda manejar. Antes de enviar un nuevo frame al cliente, el sistema verifica si el cliente está listo para recibirla, inspeccionando si la cola de mensajes del WebSocket está llena. Si la cola está llena, el sistema espera brevemente y verifica nuevamente, en un ciclo, hasta que el cliente esté listo para recibir más datos. Este enfoque preventivo evita sobrecargar al cliente con más datos de los que puede procesar, manteniendo la estabilidad de la conexión y asegurando una entrega de datos más coherente y fluida. Esto es particularmente vital en aplicaciones de streaming de video para evitar el lag y asegurar una visualización fluida y de alta calidad en el cliente.

```

while (true)
{
    // Obtener un puntero al cliente del WebSocket.
    // 'wsCamera' es una instancia del WebSocket para la cámara, y 'client(cameraClientId)'
    // devuelve un puntero al cliente WebSocket con el ID 'cameraClientId'.
    AsyncWebSocketClient * clientPointer = wsCamera.client(cameraClientId);

    // Verificar si el cliente no está conectado (no es NULL) o si la cola de
    // mensajes del cliente no está llena.
    // Si alguna de las condiciones es verdadera, se sale del bucle con 'break'.
    if (!clientPointer || !(clientPointer->queueIsFull()))
    {
        break;
    }

    // Pequeño retraso antes de la próxima iteración del bucle.
    delay(1);
}

```

Una vez desarrollada la aplicación y validado su funcionamiento, la siguiente característica a evaluar fue a qué red se puede conectar el ESP32 se testearon los siguientes dos escenarios

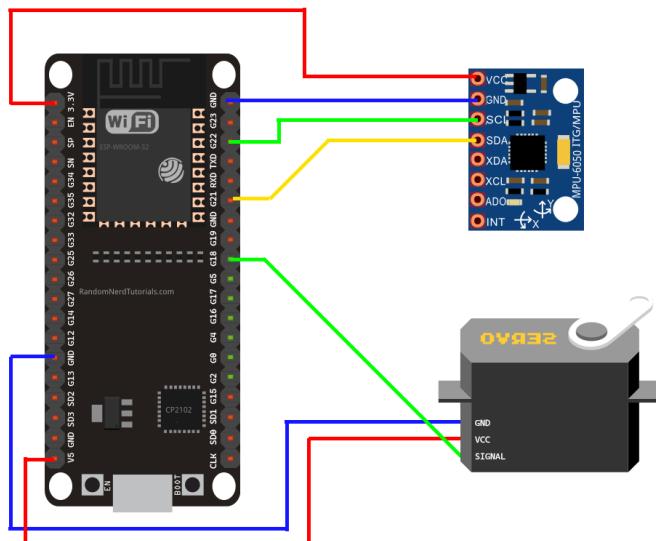


Figura C3 - Circuito MPU y Servo

Con respecto al código del programa, el mismo resultó ser prácticamente el mismo, la mayoría de los cambios fueron referidos a la adhesión de la librería necesaria para mover los servos, así como las órdenes para realizar esta tarea. Lo principal para que el movimiento se realizara tuvo que ver con el tipo de dato que guardábamos de rotación, el mismo podía incluir números decimales, mientras que el servo solo permite recibir enteros, por esto mismo se agregó

una instrucción para redondear el dato. A continuación se puede observar la sección del código que realiza lo previamente mencionado.

```
float dt = 0.05;
gyroX = trunc(gyroX); // Se almacenan únicamente 2 dígitos
rotation += gyroX * dt;

int pos_servo = rotation * -90; // Se mapea el dato ya que oscila entre 0 y -1
myservo.write(pos_servo);
```

A partir de realizar todas estas modificaciones el programa de prueba funcionó como se esperaba, a medida que se movía el sensor, giraba el servo. En la sección de Documentación Relacionada se podrá encontrar un video del funcionamiento, junto con el código completo, en caso de que desee revisarse.

Sin embargo, prueba mediante, surgieron algunos problemas con la medición del giroscopio. Tras un tiempo de ejecución del programa, observamos que el punto de referencia a partir del cual se calcula la rotación se traslada. Afortunadamente, luego de investigar, llegamos a la conclusión de que puede tratarse de un problema de calibración, ya que el sensor puede tener algún error de medición dependiendo del lugar geográfico donde se esté usando.

#### C.4 Prueba inicial: Uso del ESP32 Como Access Point

Para utilizar el ESP32 como Access Point se debe realizar la configuración correspondiente. En primer lugar, se setearon las credenciales de la WiFi que creará el AP de la placa

```
const char* ssid      = "NowISeeYou";
const char* password = "12345678";
```

Y luego en el setup de la placa, se debe levantar el servicio de Access Point

```
// Create an access point in the ESP32
WiFi.softAP(ssid, password);
IPAddress IP = WiFi.softAPIP();
Serial.print("AP IP address: ");
Serial.println(IP);
```

En este caso, como los hosts se conectan a un webserver que se encuentra en el mismo hardware físico que el Access point, los tiempos de latencia entre la comunicación de la red con el servidor web serán muy bajos.

Esta configuración permite establecer una calidad de imagen superior sin perder tantos frames generando un stream más fluido a una mejor resolución. La demostración en video de esta arquitectura se encuentra en la sección de “Documentación Relacionada”

## C.5 Uso del ESP32 conectándose a una red Externa

Para conectar el ESP32 a una red externa, se debe realizar la configuración correspondiente. En primer lugar, se setearon las credenciales de la WiFi a la que nos queremos conectar.

```
const char* ssid      = "WiFi Name";
const char* password = "password";
```

Y luego en el setup de la placa, se debe establecer la conexión a la WiFi. En este código, la placa no continua con su proceso de setup hasta no conectarse a la red

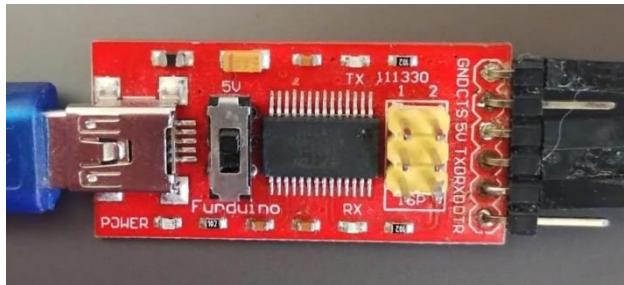
```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
}
Serial.println("Connected to WiFi");
Serial.print("IP Address: ");
Serial.println(WiFi.localIP());
```

En este caso como el webserver y el Access point se encuentran en distinto lugar físico, los tiempos de latencia entre la comunicación de la red con el servidor web serán más altos. La demostración en video de esta arquitectura se encuentra en la sección de “Documentación Relacionada”

## C.6 Prueba de funcionamiento de los ESP32-CAM

La cátedra dispuso de 3 MCU ESP32-CAM de los cuales nos advirtió que uno de los tres no funcionaba. Tras realizar múltiples pruebas, se llegó a la conclusión de cuál era el problemático. En esta sección revisaremos la metodología llevada a cabo.

A la hora de programar un ESP32-CAM es necesario la utilización de hardware externo a la placa. Para ello existen dos opciones principales: el programador ESP32-CAM-MB (Figura 13), y el programador que se puede ver en la Figura 14.



**Figura C4** - Programador de ESP CAM



**Figura C5** - Programador de ESP CAM

Una de las ESP era nueva y vino con su propio programador como el de la Figura 13, es por esto que lo primero que hicimos fue probar las 3 ESP con dicho programador. El resultado fue peor del esperado, pues únicamente la ESP nueva funcionó con el mismo. En la Figura 14 se puede ver el error que surgía en la consola del IDE de Arduino al intentar cargar un programa en ambas ESP.

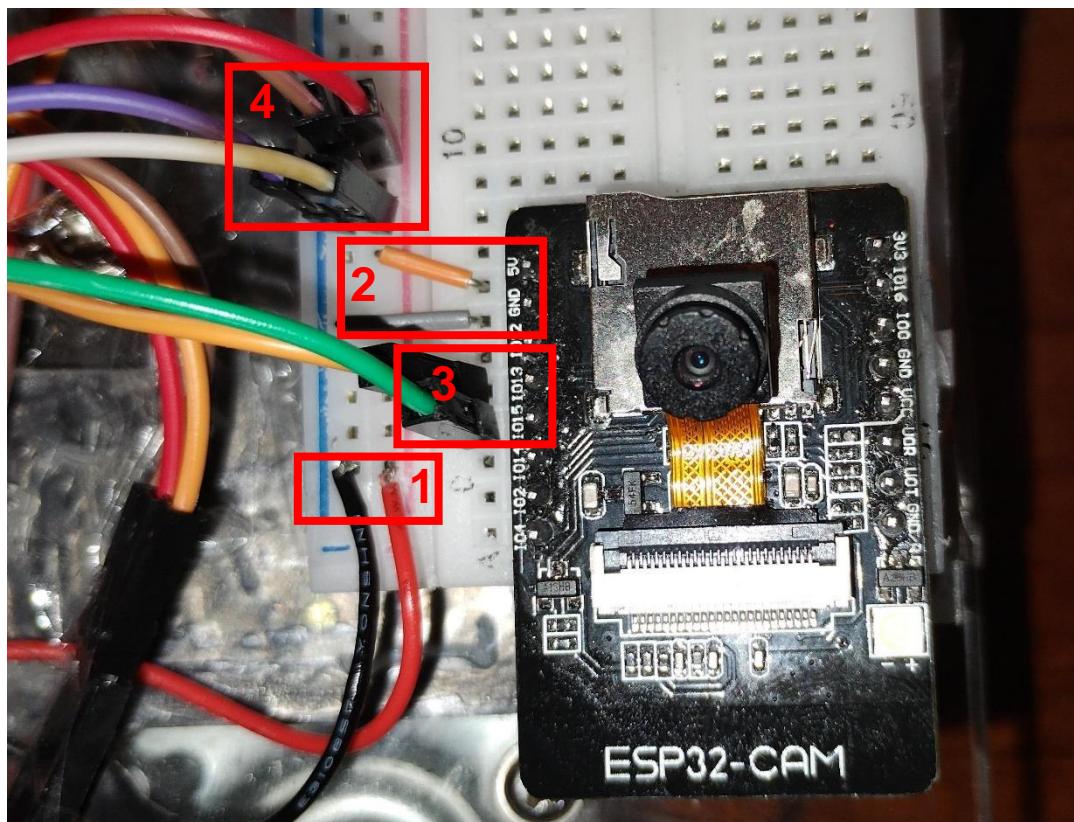
```
Output Serial Monitor
Sketch uses 1506869 bytes (47%) of program storage space. Maximum is 3145728 bytes.
Global variables use 70216 bytes (21%) of dynamic memory, leaving 257464 bytes for local variables. Maximum is 327680 bytes.
esptool.py v4.5.1
Serial port COM4
Connecting.....A fatal error occurred: Failed to connect to ESP32: No serial data received.
For troubleshooting steps visit: https://docs.espressif.com/projects/esptool/en/latest/troubleshooting.html
Failed uploading: uploading error: exit status 2
```

**Figura C6** - Error al cargar un programa en las ESP32-CAM

Al comentarle esto a la cátedra, se nos dio para leer un trabajo de años anteriores en el que conectaban el programador de la Figura 13 y explicaban cómo hacerlo. Con el cambio de programador, logramos cargar programas en una de las dos ESP32-CAM “viejas”, tal como se ve en la Figura 15. Se concluye entonces que una de las dos ESP32-CAM no tiene problemas en la placa, sino que la misma no es compatible con el programador de la Figura 14.

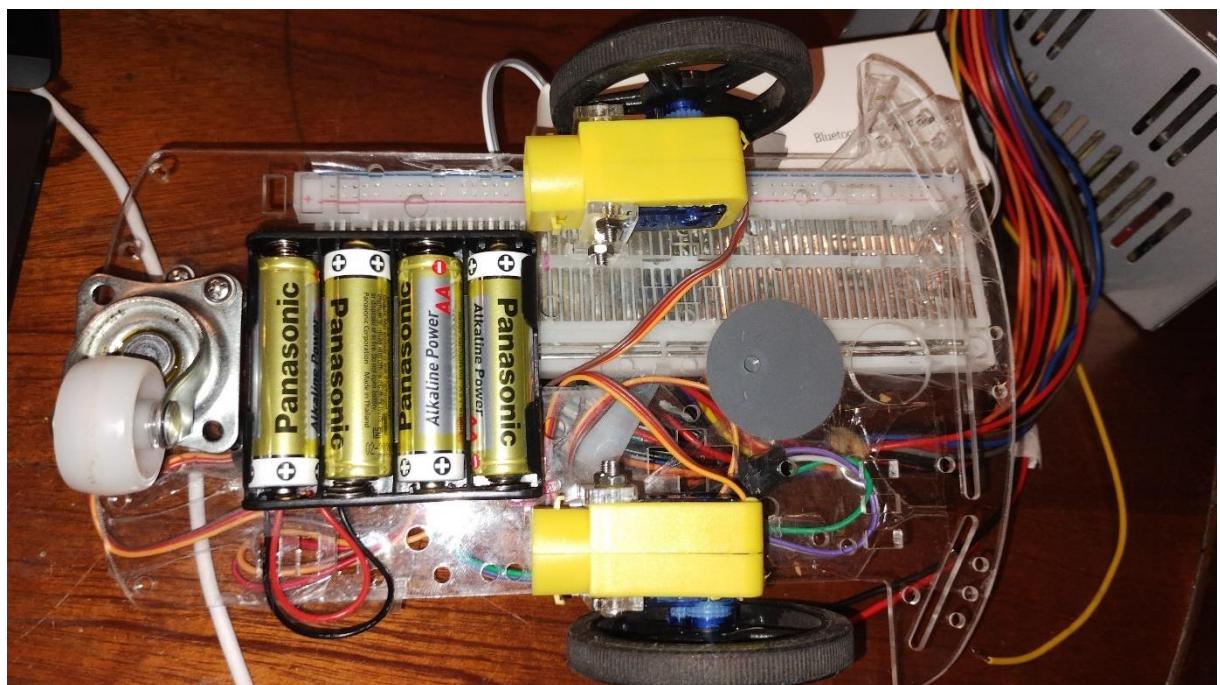
## C.7 Prototipo: Conexionado físico del vehículo para lograr únicamente su movimiento utilizando MQTT

En la Figura 24 se puede ver el conexionado físico del auto.



**Figura C7** – Conexionado físico de la ESP32-CAM que controla el movimiento del auto

En los pines 1 se conecta la alimentación para la cual se dispone de 4 pilas tipo AA que se encuentran ubicadas en la parte inferior de la estructura, tal como se puede ver en la Figura 25.



**Figura C8** – Alimentación del autito

Luego, el grupo de pines 2 es el encargado de alimentar la ESP32-CAM quien, mediante sus pines 13 y 15, controla los servos (en la imagen es el grupo de pines 3). Finalmente, los grupos de pines 4 y 5 son los que alimentan los servomotores.

Es importante aclarar que el conexiónado actual fue diseñado para lograr la funcionalidad esperada, pero está alejado de lo que va a ser la versión final, ya que se espera que este mejore en prolijidad y geometría conforme el proyecto se acerca a las etapas finales.

## C.8 Configuración del Flask Web Server

```
5.  from flask import Flask, render_template
6.  app = Flask(__name__)
7.
8.  @app.route('/')
9.  def index():
10.     return render_template('index.html')
11.
12. if __name__ == '__main__':
13.     app.run(host='0.0.0.0', port=5000)
```

## C.9 Código del publicador MQTT al auto

```
client = mqtt.Client()
client.connect("192.168.129.132", 1883, 60)

while True:
    c = stdscr.getch()

    if c != -1:
        if c == curses.KEY_UP:
            last_key_pressed = "U"
            client.publish("test", "U")
        elif c == curses.KEY_DOWN:
            last_key_pressed = "D"
            client.publish("test", "D")
        elif c == curses.KEY_RIGHT:
            last_key_pressed = "R"
            client.publish("test", "R")
        elif c == curses.KEY_LEFT:
            last_key_pressed = "L"
            client.publish("test", "L")
        elif c == ord('q'): # Press 'q' to quit
            break
        else: # No key was pressed
            if (last_key_pressed != "S"):
                client.publish("test", "S")
    last_key_pressed = "S"
```

## C.10 Código del suscriptor desde la ESP32-CAM que controla el auto

```
// Instrucciones para instanciar al cliente de MQTT
WiFiClient espClient;
PubSubClient client(espClient);

// Instrucciones para establecer la conexión MQTT
client.setServer("192.168.129.132", 1883);
client.setCallback(callback);
if (client.connect("ESP32Client")) {
    Serial.println("connected");
    Serial.println(client.subscribe("test"));
}
```

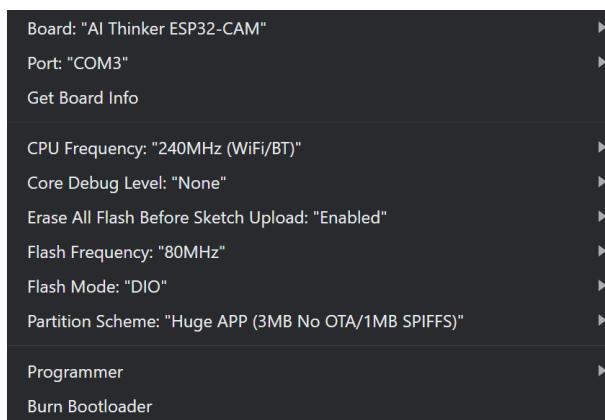
## C.11 Código de la función recall para el control del auto

```
if ((char)message[0] == 'U') {
    // Forward
    servo_left.writeMicroseconds(1700);
    servo_right.writeMicroseconds(1280);
}
else if ((char)message[0] == 'D') {
    // Backwards
    servo_left.writeMicroseconds(1280);
    servo_right.writeMicroseconds(1700);
}
else if ((char)message[0] == 'L') {
    // Turn left
    servo_left.writeMicroseconds(1480);
    servo_right.writeMicroseconds(1280);
}
else if ((char)message[0] == 'R') {
    // Turn right
    servo_left.writeMicroseconds(1700);
    servo_right.writeMicroseconds(1480);
}
else if ((char)message[0] == 'S'){
    servo_left.writeMicroseconds(1480);
    servo_right.writeMicroseconds(1480);
}
```

## C.11 Para cargar el código a la placa ARduino

Se utilizó el driver CH341SER para la placa ESP32 Wrover Module el cual se puede descargar desde múltiples plataformas digitales.

Se utilizó la board manager esp32 de espressif en la versión 2.0.11



**Figura C9** – Alimentación del autito

Bibliotecas instaladas en Arduino

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <ESP32Servo.h>
#include <WiFiUdp.h>
```