

# Análisis e Implementación de una Red CAN Virtual usando Docker y SocketCAN - (CAN) BUS - Físico

Mateo Eduardo Bermeo Pesántez

*Facultad de Ingeniería - Telecomunicaciones*

*Universidad de Cuenca*

Cuenca, Ecuador

[mateo.bermeop@ucuenca.edu.ec](mailto:mateo.bermeop@ucuenca.edu.ec)

Santiago Andrés Guillén Malla

*Facultad de Ingeniería - Telecomunicaciones*

*Universidad de Cuenca*

Cuenca, Ecuador

[santiago.guillenm@ucuenca.edu.ec](mailto:santiago.guillenm@ucuenca.edu.ec)

Vicente Paúl Jiménez Ávila

*Facultad de Ingeniería - Telecomunicaciones*

*Universidad de Cuenca*

Cuenca, Ecuador

[paul.jimeneza@ucuenca.edu.ec](mailto:paul.jimeneza@ucuenca.edu.ec)

**Resumen**—Este informe presenta el desarrollo completo del Taller 4, dividido en dos partes: una red CAN virtual y una red CAN física. En la primera, se implementó y analizó una red CAN simulada mediante herramientas de código abierto en un entorno Linux con Docker, utilizando la pila SocketCAN, los comandos de `can-utils` y los programas SavvyCAN e ICSim para la visualización de tramas. En la segunda parte, se estableció la comunicación física entre módulos MCP2515 y Arduinos, así como la captura y análisis del tráfico mediante un adaptador USB-CAN y Wireshark.

El objetivo general de la práctica fue comprender el funcionamiento del protocolo CAN tanto en un entorno simulado como físico, identificar las diferencias entre tramas estándar y extendidas, y aplicar las herramientas de análisis de tráfico para interpretar el intercambio de datos en una red automotriz real y virtual.

**Index Terms**—CAN Bus, SocketCAN, Docker, SavvyCAN, ICSim, MCP2515, Wireshark, Arduino

## I. INTRODUCCIÓN

La red CAN (Controller Area Network) es un protocolo de comunicación ampliamente utilizado en la industria automotriz y sistemas embebidos, caracterizado por su robustez, eficiencia y capacidad de comunicación distribuida entre múltiples nodos. En este taller se desarrolla un entorno completamente virtual para experimentar con este protocolo, evitando el uso de hardware físico mediante la emulación de interfaces CAN.

El entorno de trabajo se implementó en una máquina virtual con Ubuntu 22.04, sobre la cual se configuró un contenedor Docker con las herramientas necesarias para emular una red CAN. Se utilizó SocketCAN, el conjunto de drivers y utilidades nativas de Linux para trabajar con CAN, junto con SavvyCAN e ICSim para la visualización y simulación gráfica de tramas CAN.

La práctica se divide en tres partes: en la primera se construye el entorno de comunicación virtual, en la segunda se utiliza la herramienta SavvyCAN para el análisis visual de mensajes, y en la tercera se emplea el simulador ICSim para recrear un tablero automotriz virtual y analizar su tráfico.

Además del entorno virtual, se desarrolló una segunda fase orientada a la implementación física del bus CAN, empleando módulos MCP2515 acoplados a placas Arduino y un adaptador USB2CAN conectado al host Ubuntu. Esta parte permitió contrastar el comportamiento real del bus frente a la simulación, observando cómo las tramas transmitidas y recibidas mantienen la misma estructura definida por el estándar ISO 11898.

Durante las pruebas físicas se configuró el bus a 125 kbps, se verificó el funcionamiento de tramas estándar y extendidas, y se analizó el tráfico con herramientas como `candump` y Wireshark. Esto permitió consolidar los conocimientos adquiridos en la parte virtual, validando el proceso completo de comunicación CAN desde la generación, transmisión, captura y decodificación de tramas en un entorno real.

De esta manera, el taller integró tanto los conceptos teóricos como la aplicación práctica del protocolo CAN, desde la emulación en software hasta su despliegue sobre hardware real.

## II. MARCO TEÓRICO

### II-A. Protocolo CAN

El protocolo CAN (Controller Area Network) fue desarrollado por Bosch en 1986 para permitir la comunicación confiable entre microcontroladores sin necesidad de una computadora central [1]. Su principal característica es el mecanismo de arbitraje por prioridad, donde el identificador (ID) determina

qué nodo gana el acceso al bus [2]. El estándar define tramas de tipo estándar (11 bits) y extendida (29 bits), con hasta 8 bytes de datos [3].

## *II-B. SocketCAN en entornos Linux*

**SocketCAN** es la implementación nativa del protocolo CAN en el kernel de Linux, desarrollada por Volkswagen Research y actualmente mantenida por la comunidad [4]. Permite la comunicación con buses CAN físicos o virtuales mediante interfaces de red (can0, vcan0) y herramientas como cansend, candump y cangen, facilitando la emulación y análisis de tramas en entornos controlados.

## *II-C. Módulo MCP2515 y aplicaciones de simulación*

El controlador **MCP2515** de Microchip es un dispositivo autónomo que implementa la capa de enlace y física del protocolo CAN, comunicándose con microcontroladores vía SPI [5]. Junto con el transceptor TJA1050, permite crear redes físicas con bajo costo. Por su parte, las herramientas **SavvyCAN** e **ICSim** proporcionan interfaces gráficas para visualizar, reproducir y simular tráfico automotriz, siendo ampliamente utilizadas en entornos educativos y de desarrollo.

### III. DESARROLLO

*Taller CAN Virtual*

### *III-A. Parte 1: Creación de la interfaz virtual y pruebas con SocketCAN*

Para iniciar la práctica, se creó una interfaz virtual de tipo vcan directamente sobre el sistema host (Ubuntu). Esta interfaz permite simular una red CAN sin necesidad de hardware físico.

```
sudo ip link add dev vcan0 type vcan
sudo ip link set can0 up
ip add
```

La ejecución de los comandos anteriores permitió verificar la creación de la interfaz `vcan0` (Figura 1). Posteriormente, se habilitó el acceso gráfico desde el entorno de Docker para mostrar interfaces visuales con el comando:

```
xhost +local:root
```

```
[root@mlnx-0 ~]# mif -l /etc/mif/mif.conf & lpm add
1: los :<OLBPACK>,UP,LOWER_UP mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
    inet6 ::/128 scope host
        valid_lft forever preferred_lft forever
2: em0p3 :<BROADCAST,MULTICAST>,UP,LOWER_UP mtu 1500 qdisc mq state UNKNOWN group default qlen 1000
link/ether 00:0c:29:00:00:03 brd ff:ff:ff:ff:ff:ff
inet 10.0.2.1/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
    valid_lft 84863sec
    inet6 fd17:025e::f037:2334:70ff:fe93:7004/64 scope global temporary dynamic
        valid_lft 2592000sec
        inet6 fe80::f037:2334%enp0s3/64 scope link
            valid_lft 2592000sec
            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic mgntmpaddr noprefixroute
                valid_lft 8008sec
                inet6 fe80::d609:4ff%enp0s3/64 scope link
                    valid_lft 2592000sec
                    inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                        valid_lft 8008sec
                        inet6 fe80::d609:4ff%enp0s3/64 scope link
                            valid_lft 2592000sec
                            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                valid_lft 8008sec
                                inet6 fe80::d609:4ff%enp0s3/64 scope link
                                    valid_lft 2592000sec
                                    inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                        valid_lft 8008sec
                                        inet6 fe80::d609:4ff%enp0s3/64 scope link
                                            valid_lft 2592000sec
                                            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                valid_lft 8008sec
                                                inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                    valid_lft 2592000sec
                                                    inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                        valid_lft 8008sec
                                                        inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                            valid_lft 2592000sec
                                                            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                                valid_lft 8008sec
                                                                inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                                    valid_lft 2592000sec
                                                                    inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                                        valid_lft 8008sec
                                                                        inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                                            valid_lft 2592000sec
                                                                            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                                                valid_lft 8008sec
                                                                                inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                                                    valid_lft 2592000sec
                                                                                    inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                                                        valid_lft 8008sec
                                                                                        inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                                                            valid_lft 2592000sec
                                                                                            inet6 fd17:025e::f037:2334:b330:ed2ce/64 scope global dynamic noprefixroute
                                                                                                valid_lft 8008sec
                                                                                                inet6 fe80::d609:4ff%enp0s3/64 scope link
                                                                
3: docker0 :<NO-CARRIER,BROADCAST,MULTICAST>,UP,mtu 1500 qdisc noqueue state DOWN group default
link/ether 00:0c:29:00:00:04 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
4: vcan0 :<NO-CARRIER>,UP,LOWER_UP mtu 72 qdisc noqueue state UNKNOWN group default qlen 1000
link/vcan0
```

Figura 1: Interfaz virtual vcan0 creada en el host Ubuntu.

Luego, se creó un directorio de trabajo y un archivo Dockerfile que define el entorno del laboratorio:

```
mkdir ~/taller can
```

```
cd ~/taller_can  
nano Dockerfile
```

El contenido del archivo Dockerfile fue el siguiente:

```
FROM ubuntu:22.04
ENV DEBIAN_FRONTEND=noninteractive
WORKDIR /root

RUN apt update && apt upgrade -y && \
    apt install -y iproute2 net-tools
    can-utils git nano
```

```
CMD modprobe vcan || true && \
    ip link add dev vcan0 type vcan || \
    true && \
    ip link set up vcan0 && \
    bash
```

Con este archivo se construyó la imagen base del taller:

```
docker build -t taller_can .
```

Una vez finalizada la compilación, se ejecutó el contenedor con permisos privilegiados y soporte gráfico:

```
docker run -it --name can_lab_parte1 \
--privileged \
--network host \
-e DISPLAY=$DISPLAY \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-v /home/paul/taller_can:/root/work \
taller_can bash
```

Dentro del contenedor (Figura 2), se comprobó la interfaz CAN disponible con:

```
ip link show  
ifconfig vcan0
```

Figura 2: Visualización de la interfaz `vcan0` dentro del contenedor.

A continuación, se probaron las herramientas básicas de can-utils. Primero se generó tráfico aleatorio en la interfaz:

cangen vcan0

En otra terminal (Figura 3) del mismo contenedor se monitoreó el tráfico con:

```
docker exec -it can lab bash
```

candump vcan0

```
root@nietzsche:~# docker exec -it can_lab bash
permission denied while trying to connect to the socket '/var/run/docker.sock' (errno 13)
10.0.0.100:~/can_lab$ docker exec -it can_lab bash
[sudo] paraexecs para-pais:
root@ccfc913be23:~# root@ccfc913be23:~# cat /proc/meminfo | grep vcan
vcan      404 [8]    95 BB 2C 9F 9F D4 SE
vcan      334 [8]    95 BB 2C 9F 9F D4 SE
vcan      334 [8]    F0 15 AD 60 A3 7D 67
vcan     330 [8]    05 00 74 4E B6 FC 54
vcan      330 [8]    05 00 74 4E B6 FC 54
vcan      664 [8]    F1 86 90 43 A8 A5 31
vcan      468 [8]    F1 86 90 43 A8 A5 31
vcan      468 [8]    4A 13 E9 7B 84 81 B1 72
vcan     167 [8]    A3 E9 55 5B 6B 89 7C
vcan     167 [8]    A3 E9 55 5B 6B 89 7C
vcan     170 [8]    A8 82 57 8A 5A 56
vcan     170 [8]    A8 82 57 8A 5A 56
vcan     766 [8]    A4 41 87 1C 5A 56
vcan     766 [8]    A4 41 87 1C 5A 56
vcan     315 [8]    00 00 00 00 00 00 00 00
vcan     315 [8]    00 00 00 00 00 00 00 00
vcan     613 [8]    89 E1 00 00 00 00 00 00
vcan     613 [8]    89 E1 00 00 00 00 00 00
vcan     508 [4]    ED AD 36 13
vcan     508 [4]    ED AD 36 13
vcan     173 [2]    00 00 00 00 00 00 00 00
vcan     173 [2]    00 00 00 00 00 00 00 00
vcan     682 [5]    18 BB 61 1F 41
vcan     682 [5]    18 BB 61 1F 41
vcan     340 [8]    58 6F 3F 50 50 79 01 42
root@ccfc913be23:~# dmesg vcan0
root@ccfc913be23:~# ifconfig vcan0
vcan0: flags=1390<UP,BROADCAST,NOARP mtu 72
        txqueuelen 1000
        RX: packets 0 bytes 0 (0.0 B)
        TX: packets 0 bytes 0 (0.0 B)
        errors 0 dropped 0 overrun 0 collisions 0
(UNSPEC)
        Rx packets 0 bytes 0 (0.0 B)
        Tx packets 0 bytes 0 (0.0 B)
        Tx errors 0 dropped 0 overrun 0 carrier 0 collisions 0
root@ccfc913be23:~# cat /proc/meminfo | grep vcan
vcan      404 [8]    95 BB 2C 9F 9F D4 SE
vcan      334 [8]    95 BB 2C 9F 9F D4 SE
vcan      334 [8]    F0 15 AD 60 A3 7D 67
vcan     330 [8]    05 00 74 4E B6 FC 54
vcan      330 [8]    05 00 74 4E B6 FC 54
vcan      664 [8]    F1 86 90 43 A8 A5 31
vcan      468 [8]    F1 86 90 43 A8 A5 31
vcan      468 [8]    4A 13 E9 7B 84 81 B1 72
vcan     167 [8]    A3 E9 55 5B 6B 89 7C
vcan     167 [8]    A3 E9 55 5B 6B 89 7C
vcan     170 [8]    A8 82 57 8A 5A 56
vcan     170 [8]    A8 82 57 8A 5A 56
vcan     766 [8]    A4 41 87 1C 5A 56
vcan     766 [8]    A4 41 87 1C 5A 56
vcan     315 [8]    00 00 00 00 00 00 00 00
vcan     315 [8]    00 00 00 00 00 00 00 00
vcan     613 [8]    89 E1 00 00 00 00 00 00
vcan     613 [8]    89 E1 00 00 00 00 00 00
vcan     508 [4]    ED AD 36 13
vcan     508 [4]    ED AD 36 13
vcan     173 [2]    00 00 00 00 00 00 00 00
vcan     173 [2]    00 00 00 00 00 00 00 00
vcan     682 [5]    18 BB 61 1F 41
vcan     682 [5]    18 BB 61 1F 41
vcan     340 [8]    58 6F 3F 50 50 79 01 42
```

Figura 3: Tramas CAN aleatorias observadas con candump.

También se guardó el tráfico en un archivo y se reprodujo posteriormente:

```
candump vcan0 > /root/can_traffic.log  
canplayer -I /root/can_traffic.log
```

Luego se enviaron tramas personalizadas con datos en formato hexadecimal:

```
cansend vcan0 012#686F6C616D756E646F
```

Esta trama corresponde al mensaje “holamundo”. El resultado fue visible en la terminal de `candump`, confirmando la correcta transmisión (Figura 4).

```
root@ecocel:~# cd /tmp
root@ecocel:~/tmp# ./cansend
root@ecocel:~/tmp# ./cansend
root@ecocel:~/tmp# ./cansend
root@ecocel:~/tmp# ./cansend
```

Figura 4: Envío y recepción del mensaje holamundo en formato hexadecimal.

Finalmente, se probaron tramas estándar, extendidas y RTR (Figura 5):

```
cansend vcan0 123#1122334455667788  
cansend vcan0 12345678#1122334455667788  
cansend vcan0 123#R
```

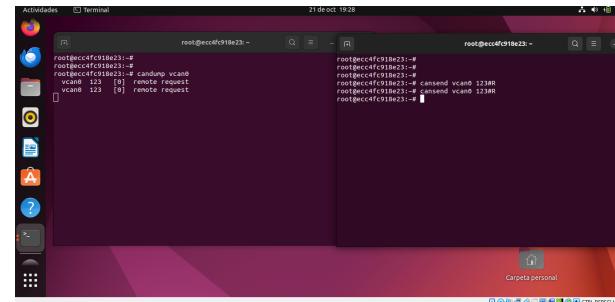


Figura 5: Ejecución de tramas estándar, extendidas y RTR (bit RTR=1).

Para el análisis de tráfico con Wireshark, se ejecutó el generador de tramas en una terminal y Wireshark en el host:

```
# En el contenedor:  
cangen vcan0
```

```
# En el host:  
sudo wireshark &
```

Seleccionando la interfaz vcan0, se visualizaron las tramas CAN generadas en tiempo real (Figura 6).

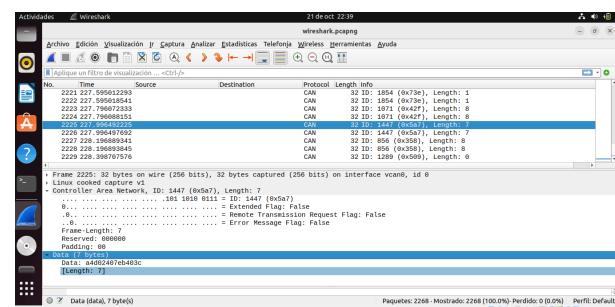


Figura 6: Captura de tramas CAN en Wireshark.

### *III-B. Parte 2: Visualización de tramas con SavvyCAN*

En la segunda parte del taller se utilizó la aplicación Savvy-CAN para analizar gráficamente las tramas CAN generadas. Primero se verificó la conexión gráfica del contenedor con el host (Figura 7):

```
echo $DISPLAY  
export DISPLAY=:0  
apt update && apt install -y x11-apps  
xeyes &  
xclock &
```

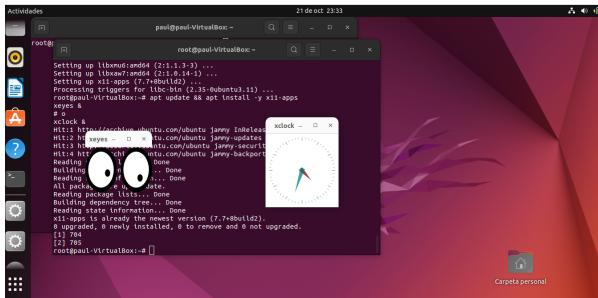


Figura 7: Prueba del entorno gráfico del contenedor mediante X11.

Posteriormente se instalaron las dependencias necesarias para compilar SavvyCAN (Figura 8):

```
apt update && apt install -y \
build-essential git curl \
qtbase5-dev qtchooser qt5-qmake qtbase5-
dev-tools \
qttools5-dev qttools5-dev-tools \
qtdeclarative5-dev qml-module-qtquick-
controls \
qml-module-qtquick-controls2 qml-module-
qtquick-dialogs \
qml-module-qtquick-layouts qml-module-
qtquick-window2 \
qml-module-qtgraphicaleffects qml-module-
qtquick2 \
libqt5serialbus5-dev libqt5serialport5-
dev \
libqt5svg5-dev libqt5charts5-dev
```

Luego se descargó y compiló SavvyCAN dentro del contenedor:

```
cd /root
git clone
https://github.com/collin80/SavvyCAN.git
cd SavvyCAN
qmake
make -j$(nproc)
```

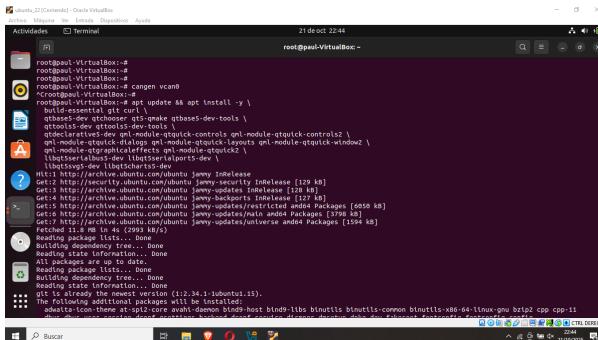


Figura 8: Compilación de SavvyCAN dentro del contenedor.

Una vez compilado, se ejecutó SavvyCAN con soporte gráfico:

```
export DISPLAY=:0
cd /root/SavvyCAN
./SavvyCAN &
```

En la interfaz de SavvyCAN se añadió una nueva conexión:

- **Connections** → Add New Device Connection
- **Connection Type:** SocketCAN
- **Interface Name:** vcan0

Tras conectar, el programa mostró el flujo de tramas CAN en tiempo real (Figura 9).

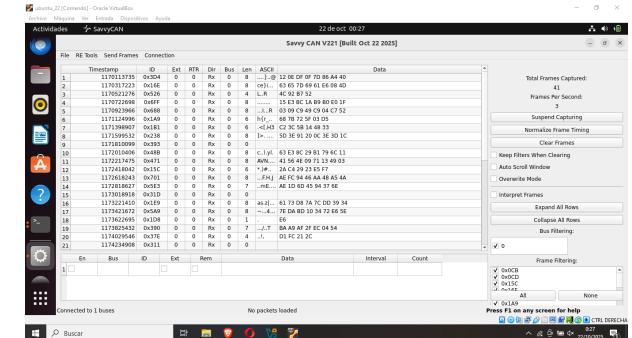


Figura 9: Conexión de SavvyCAN con la interfaz vcan0.

También se registró el tráfico en un archivo de log:

```
candump vcan0 > /root/can_traffic.log
cangen vcan0
```

El archivo generado fue abierto dentro de SavvyCAN, mostrando la misma secuencia de tramas reproducidas desde el log (Figura 10).

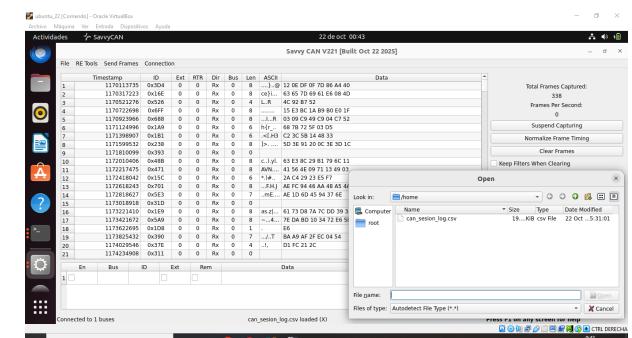


Figura 10: Visualización de un archivo can\_traffic.log dentro de SavvyCAN.

### III-C. Parte 3: Simulación con ICSim

Finalmente, se incorporó la herramienta ICSim para simular un tablero automotriz y generar tráfico CAN dinámico. Primero se instalaron las dependencias necesarias y se descargó el repositorio (Figura 11):

```
sudo apt install libsdl2-dev libsdl2-image-
```

```

dev \
    build-essential git -y
git clone
    https://github.com/zombieCraig/ICSim.git
cd ICSim
make

```

```

root@paul-VirtualBox:~# cd ICSim
root@paul-VirtualBox:~/ICSim# make
gcc -I/usr/include/SDL2 -Wall -Wextra -c -o tcsim.o tcsim.c
gcc -I/usr/include/SDL2 -Wall -Wextra -o tcsim tcsim.c libb.o -lSDL2 -lSDL2_image
gcc -I/usr/include/SDL2 -Wall -Wextra -c -o controls.o controls.c
gcc -I/usr/include/SDL2 -Wall -Wextra -o controls controls.c -lSDL2 -lSDL2_image

```

Figura 11: Compilación exitosa de ICSim y Controls.

ICSim genera dos ejecutables: `icsim` (tablero, Figura 12) y `controls` (panel de control, Figura 13). Ambos se ejecutaron en ventanas separadas del contenedor:

```
# En la primera terminal
./icsim vcan0
```

```
# En la segunda terminal
cd ~/ICSim
./controls vcan0
```



Figura 12: Simulador automotriz (IC Simulator).

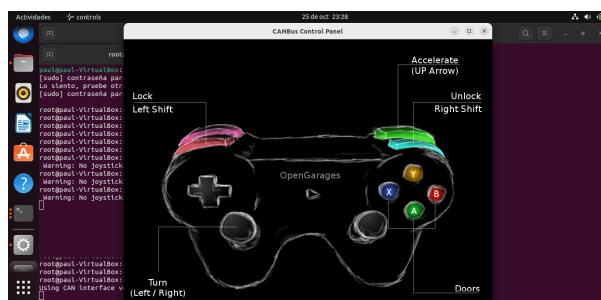


Figura 13: Simulador automotriz (CANBus Control Panel).

En una tercera terminal se capturó el tráfico generado:

```
candump vcan0
```

Con las teclas direccionales y las letras A, B, X, Y, así como Shift, se simularon acciones sobre el vehículo (luces, frenos, puertas). Cada acción se reflejaba inmediatamente en el tráfico CAN observado en la terminal (Figura 14).

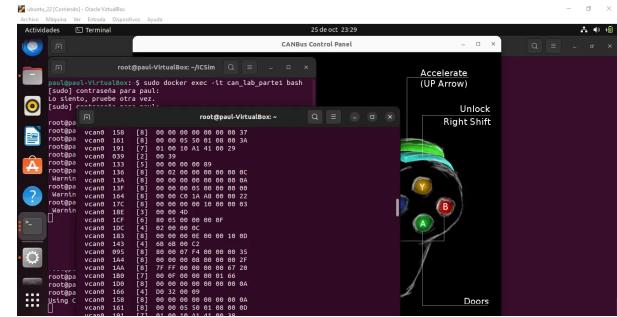


Figura 14: Captura de tramas CAN generadas por ICSim.

Estas pruebas permitieron visualizar en tiempo real la correspondencia entre eventos gráficos (puertas abiertas, direccionales en la Figura 15) y las tramas CAN transmitidas, comprendiendo así el comportamiento de un bus CAN automotriz real en un entorno virtual controlado.

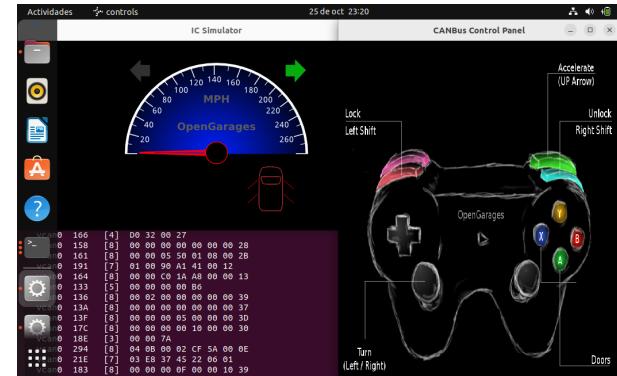


Figura 15: Simulación de puertas abiertas y direccional derecho

### Taller CAN Físico

#### III-D. Parte 1: Comunicación entre módulos CAN-Arduino

Para la implementación física del bus CAN se utilizó un adaptador USB-CAN conectado a la máquina virtual Ubuntu, junto con dos módulos Arduino equipados con controladores MCP2515 para establecer una comunicación tipo emisor-receptor.

En primer lugar, se verificó que el adaptador CAN fuese detectado correctamente por el sistema con el comando (Figura 16):

```
lsusb
```

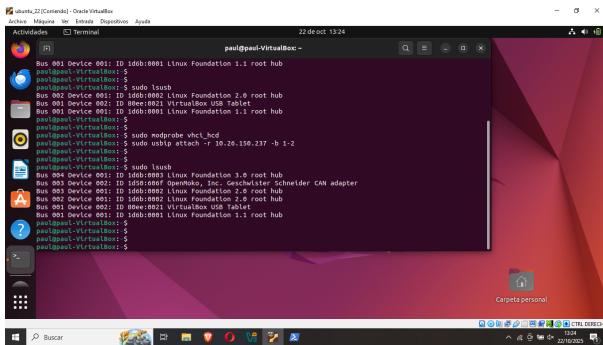


Figura 16: Verificación de la conexión del adaptador CAN mediante lsusb.

Con la conexión física establecida entre los Arduinos y los módulos MCP2515 (Figura 17), se procedió a cargar el programa transmisor (TX) en el primer Arduino, cuya lógica se detalla en el Algoritmo 1.

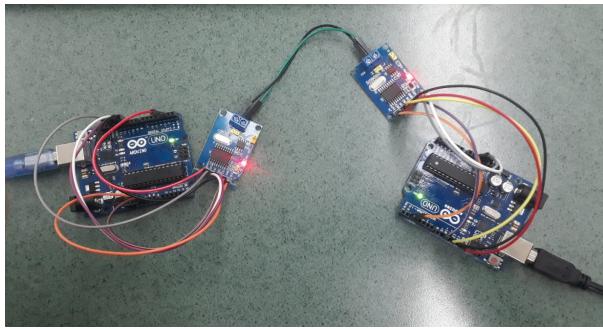


Figura 17: Conexión física entre los Arduinos y módulos MCP2515.

El código completo utilizado se incluye en el Anexo 1.

#### Algorithm 1 Transmisor CAN con Arduino

- 1: Inicializar puerto serial (9600 baudios)
- 2: Iniciar el bus CAN a 500 kbps
- 3: **if** fallo al iniciar CAN **then**
- 4:     Mostrar error y detener ejecución
- 5: **end if**
- 6: **while** activo **do**
- 7:     Construir trama estándar con ID = 0x12 y datos “hello”
- 8:     Enviar trama y esperar 1 segundo
- 9:     Construir trama extendida con ID = 0xABCD**E**F y datos “world”
- 10:    Enviar trama y esperar 1 segundo
- 11: **end while**

El código anterior transmite de forma periódica los mensajes “hello” y “world” con identificadores estándar y extendido, respectivamente.

En el segundo Arduino se cargó el programa receptor, encargado de decodificar las tramas recibidas a través del bus

CAN.

Su funcionamiento se resume en el Algoritmo 2, mientras que el código fuente completo se presenta en el Anexo 2.

---

#### Algorithm 2 Receptor CAN con Arduino

---

- 1: Inicializar puerto serial (9600 baudios)
  - 2: Iniciar el bus CAN a 500 kbps
  - 3: **if** fallo al iniciar CAN **then**
  - 4:     Mostrar error y detener ejecución
  - 5: **end if**
  - 6: **while** activo **do**
  - 7:     Leer trama del bus CAN
  - 8:     **if** existe una trama disponible **then**
  - 9:         Mostrar el tipo de trama (estándar o extendida)
  - 10:         Mostrar el ID y la longitud
  - 11:         **if** no es trama RTR **then**
  - 12:             Leer y mostrar los datos recibidos
  - 13:         **end if**
  - 14:     **end if**
  - 15: **end while**
- 

Finalmente, se verificó en los monitores seriales de ambos Arduinos el correcto envío y recepción de las tramas CAN. El transmisor mostraba la secuencia de envíos, mientras que el receptor decodificaba e imprimía las cadenas “hello” y “world” (Figura 18).

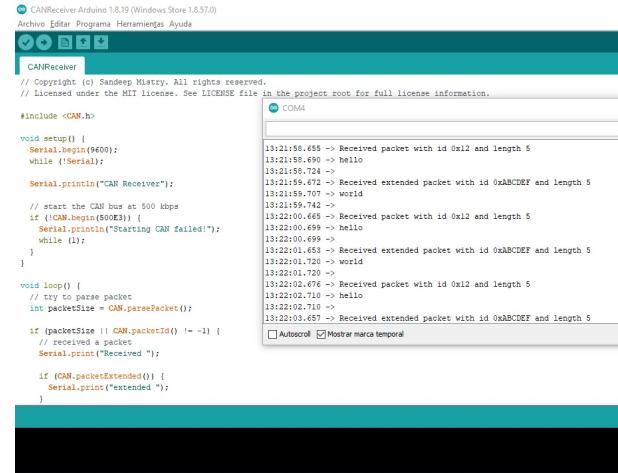


Figura 18: Recepción exitosa del mensaje “hello world” en el monitor serial.

#### III-E. Parte 2: Captura de tráfico con adaptador USB-CAN

En esta parte se utilizó el módulo MCP2515 conectado a un Arduino transmisor, y un adaptador USB2CAN como receptor conectado directamente al host Ubuntu.

La lógica del transmisor se describe en el Algoritmo 3, y su código fuente completo puede consultarse en el Anexo 3.

### Algorithm 3 Transmisor CAN con módulo MCP2515

```

1: Inicializar comunicación SPI con el módulo MCP2515
2: Configurar bus CAN a 125 kbps con cristal de 8 MHz
3: if inicio exitoso then
4:   Mostrar mensaje de inicialización correcta
5: else
6:   Mostrar error y detener ejecución
7: end if
8: while activo do
9:   Construir trama con ID = 0x123 y datos “Hello”
10:  Enviar la trama al bus
11:  if recepción disponible then
12:    Leer trama recibida e imprimir ID y datos
13:  end if
14:  Esperar 1 segundo
15: end while

```

En la Figura 19 se puede apreciar el envío de las tramas “Hello”, a través del monitor serial.

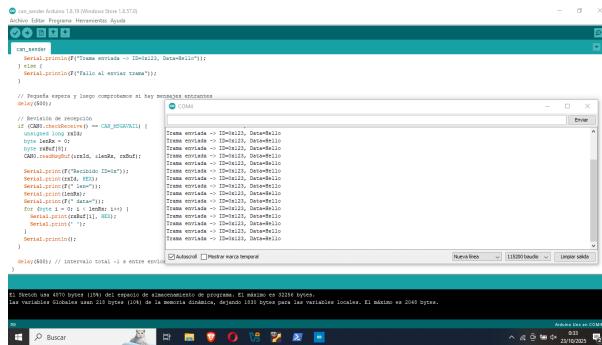


Figura 19: Monitor serial del Arduino mostrando el envío de tramas “Hello”.

Para asegurar que el dispositivo USB-CAN fuese reconocido por la máquina virtual, se configuró VirtualBox de la siguiente manera:

1. Cerrar la máquina virtual Ubuntu.
2. En la configuración de la VM, habilitar el controlador USB 3.0 (xHCI).
3. Agregar un filtro USB para el dispositivo “InnoMaker USB2CAN V3.3” o “Geschwister Schneider CAN adapter”.
4. Iniciar nuevamente la máquina virtual.

Una vez iniciada, se verificó la conexión mediante:

```
lsusb
```

y posteriormente con:

```
ip link show
```

Si la interfaz no estaba activa, se configuró y levantó manualmente con:

```

sudo apt update && sudo apt install -y
can-utils
sudo ip link set can0 type can bitrate

```

125000

```
sudo ip link set can0 up
candump can0
```

De esta manera se pudo capturar los paquetes CAN codificados en hexadecimal como “Hello” (Figura 20).

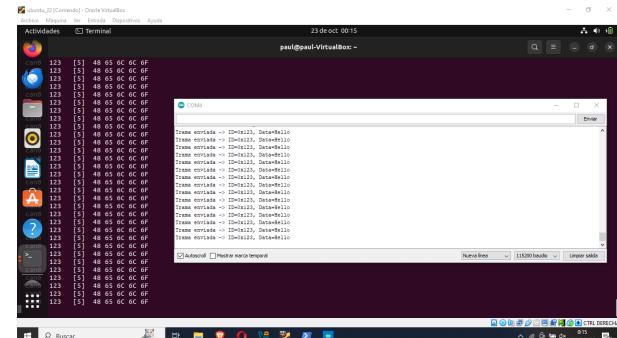


Figura 20: Captura de tráfico CAN en el host Ubuntu.

A continuación, se inició un contenedor Docker con privilegios de red y acceso al dispositivo USB, con el fin de monitorear el mismo tráfico desde un entorno aislado (Figura 21 se ve la captura simultánea):

```

sudo docker run --rm -it \
--name can_sniffer \
--network host \
--privileged \
--cap-add=NET_ADMIN \
--cap-add=NET_RAW \
-v /dev:/dev \
ubuntu:22.04 bash

```

Dentro del contenedor se instalaron las herramientas necesarias:

```
apt update && apt install -y can-utils iproute2
ip link show can0
candump can0
```

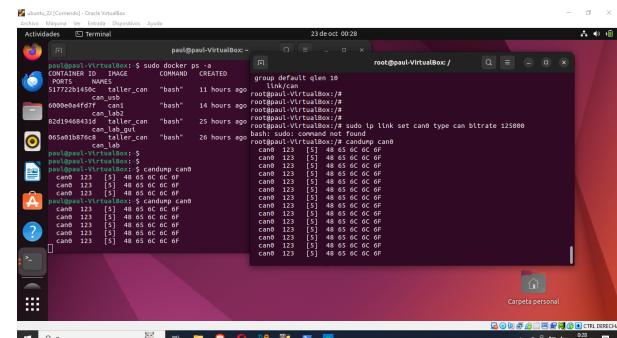


Figura 21: Captura simultánea del tráfico CAN desde el host y el contenedor Docker.

El mismo tráfico fue analizado también con Wireshark, permitiendo visualizar los campos del protocolo y el mensaje

transmitido (“Hello”). Al modificar el mensaje a “Pepito”, se observó que únicamente aparecían los cinco primeros caracteres debido a que la longitud del paquete se mantenía fija en 5 bytes (Figura 22).

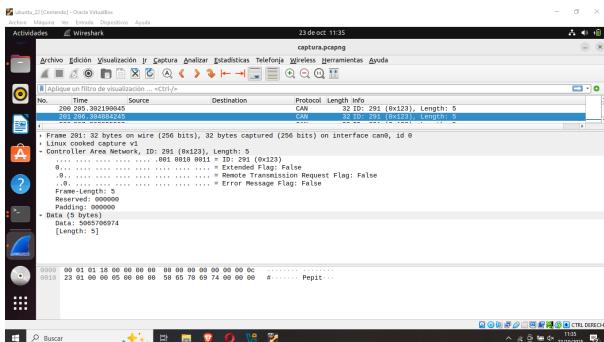


Figura 22: Análisis del mensaje truncado “Pepit” en Wireshark.

Posteriormente se analizó la diferencia entre tramas estándar y extendidas, donde se observó que las tramas extendidas poseen un campo de identificador de 29 bits y el flag “Extended” activado (Figura 23).

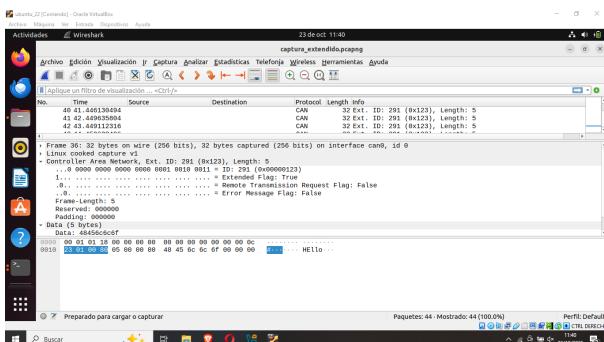


Figura 23: Visualización de trama extendida con identificador de 29 bits.

### *III-F. Comparación entre tramas estándar y extendidas*

Cuadro I: Características de la trama estándar (CAN 2.0A).

Característica	Trama Estándar (CAN 2.0A)
Identificador (ID)	11 bits
Campo IDE	Dominante (0)
Longitud de datos	0 a 8 bytes
Compatibilidad	Totalmente compatible con CAN 2.0B
Uso típico	Sensores, actuadores y redes de baja complejidad
Eficiencia	Mayor velocidad por menor sobrecarga de bits

Cuadro II: Características de la trama extendida (CAN 2.0B).

<b>Característica</b>	<b>Trama Extendida (CAN 2.0B)</b>
Identificador (ID)	29 bits
Campo IDE	Recesivo (1)
Longitud de datos	0 a 8 bytes
Compatibilidad	Requiere soporte extendido
Uso típico	Sistemas avanzados, gateways y ECUs múltiples
Eficiencia	Menor velocidad efectiva por campos adicionales

### *III-G. Preguntas de análisis*

## **1. ¿A qué se refiere el arbitraje en el contexto de las redes CAN?**

El arbitraje es el mecanismo que permite que múltiples nodos intenten transmitir simultáneamente sin colisiones. Cada trama posee un identificador (ID) que también funciona como prioridad: el nodo con el ID más bajo (más dominante) gana el acceso al bus. El proceso se realiza bit a bit, y los nodos que detectan un bit dominante distinto del transmitido ceden inmediatamente la transmisión.

**2. Explique cómo se manejan los errores durante la transmisión en el bus CAN.**

El protocolo CAN incluye detección y manejo de errores mediante varios mecanismos:

- **Error de bit:** Detecta inconsistencias entre los bits transmitidos y los leídos en el bus.
  - **Error de relleno:** Ocurre cuando no se cumple la regla de inserción de bit tras cinco bits iguales consecutivos.
  - **CRC error:** Se produce si el receptor detecta un error en el campo de verificación de redundancia cíclica.
  - **Error de forma:** Detectado si se viola la estructura de una trama.
  - **Error de ACK:** Cuando ningún nodo confirma la recepción de la trama.

Ante un error, el nodo emite una trama de error y aumenta su contador interno. Si el contador supera un umbral, el nodo pasa a modo *bus-off*, desconectándose temporalmente del bus.

3. ¿Cuál es la cantidad máxima de bits que se puede enviar en un mensaje CAN v2.0?

En una trama CAN v2.0, el campo de datos permite hasta 8 bytes, es decir, 64 bits útiles. Sin embargo, considerando todos los campos de control, delimitadores, CRC y bit stuffing, una trama completa estándar puede alcanzar aproximadamente entre 108 y 130 bits, mientras que una extendida puede llegar hasta 150 bits.

#### IV. CONCLUSIONES

La realización del taller permitió comprender de manera práctica el funcionamiento del bus de comunicación CAN tanto en entornos virtuales como físicos. A través del uso de **SocketCAN** y contenedores **Docker**, fue posible emular una red CAN completa sin requerir hardware adicional, replicando con precisión las tramas estándar, extendidas y de solicitud remota (RTR).

El uso de herramientas como `candump`, `cansend` y `cangen` facilitó la observación directa del tráfico, mientras que **Wireshark** y **SavvyCAN** brindaron una visualización detallada de los campos del protocolo, permitiendo analizar

la estructura interna de cada mensaje y su comportamiento temporal.

En la parte física, la integración de los módulos **MCP2515** con Arduinos y un adaptador **USB2CAN** permitió validar experimentalmente la comunicación real en el bus, confirmando los principios teóricos del arbitraje y la priorización por identificadores. El análisis con Wireshark corroboró la correcta recepción de los mensajes transmitidos, así como las diferencias estructurales entre tramas estándar (11 bits) y extendidas (29 bits).

El observar en el monitor serial los mensajes “hello” y “world”, y luego verlos reflejados en **Wireshark** y en el contenedor Docker, hizo que la práctica resulte especialmente interesante, ya que permitió conectar directamente la teoría con resultados tangibles.

El proceso de depurar, verificar la conexión física y finalmente observar la comunicación efectiva entre dispositivos proporcionó una comprensión mucho más clara y dinámica del funcionamiento del bus CAN.

#### REPOSITORIO DEL PROYECTO

El material completo del taller, incluyendo los códigos Arduino, Dockerfile y scripts utilizados, se encuentra disponible en: [https://github.com/santiagm/Taller4\\_Sensores](https://github.com/santiagm/Taller4_Sensores)

#### REFERENCIAS

- [1] Robert Bosch GmbH, *CAN Specification Version 2.0*, 1991, available at: <https://www.bosch-semiconductors.com/>.
- [2] Mouser Electronics, “Introduction to controller area network (can),” 2021, available at: <https://www.mouser.com/tech-articles/introduction-to-controller-area-network-can>.
- [3] International Organization for Standardization, “Iso 11898-1:2015 — road vehicles — controller area network (can),” 2015.
- [4] Linux Foundation, “Socketcan — the official can implementation in the linux kernel,” 2022, available at: <https://www.kernel.org/doc/Documentation/networking/can.txt>.
- [5] Microchip Technology Inc., *MCP2515 Stand-Alone CAN Controller with SPI Interface*, 2023, datasheet DS21801E.

#### V. ANEXOS

##### V-A. Código fuente del transmisor (Arduino CAN TX)

```

1 // Codigo del receptor
2 #include <CAN.h>
3
4 void setup() {
5   Serial.begin(9600);
6   while (!Serial);
7
8   Serial.println("CAN Receiver");
9
10 if (!CAN.begin(500E3)) {
11   Serial.println("Starting CAN failed!");
12   while (1);
13 }
14
15 void loop() {
16   int packetSize = CAN.parsePacket();
17
18   if (packetSize || CAN.packetId() != -1) {
19     Serial.print("Received ");
20     if (CAN.packetExtended())
21       Serial.print("extended ");
22     if (CAN.packetRtr()) Serial.print("RTR ");
23     Serial.print("packet with id 0x");
24     Serial.print(CAN.packetId(), HEX);
25
26     if (CAN.packetRtr()) {
27       Serial.print(" and requested length ");
28       Serial.println(CAN.packetDlc());
29     } else {
30       Serial.print(" and length ");
31       Serial.println(packetSize);
32       while (CAN.available())
33         Serial.print((char)CAN.read());
34     }
35   }
36 }
37 }
```

Listing 1: Código Arduino del transmisor CAN.

##### V-B. Código fuente del receptor (Arduino CAN RX)

```

1 #include <SPI.h>
2 #include "mcp_can.h"
3
4 const int SPI_CS_PIN = 10;
5 const int INT_PIN = 2;
6 MCP_CAN CAN0(SPI_CS_PIN);
7
8 void setup() {
9   Serial.begin(115200);
10  while (!Serial);
11  Serial.println(F("Inicializando MCP2515 (8 MHz
12    a 125 kbps..."));
13  if (CAN0.begin(MCP_ANY, CAN_125KBPS, MCP_8MHZ)
14    == CAN_OK) {
15    Serial.println(F("MCP2515 inicializado
16      correctamente."));
17    CAN0.setMode(MCP_NORMAL);
18  } else {
19    Serial.println(F("Error al iniciar
20      MCP2515."));
21    while (1);
22  }
23
24 void loop() {
25   byte data[] = { 'H', 'e', 'l', 'l', 'o' };
26   byte len = 5;
27   if (CAN0.sendMsgBuf(0x123, 0, len, data) ==
28     CAN_OK)
29     Serial.println(F("Trama enviada -> ID=0x123,
30                   Data=Hello"));
31   delay(500);
32 }
```

Listing 2: Código Arduino del receptor CAN.

##### V-C. Código fuente MCP2515

```

1 #include <SPI.h>
2 #include "mcp_can.h"
3
4 const int SPI_CS_PIN = 10;
5 const int INT_PIN = 2;
6 MCP_CAN CAN0(SPI_CS_PIN);
7
8 void setup() {
9   Serial.begin(115200);
```

```
10 while (!Serial);
11 Serial.println(F("Inicializando MCP2515 (8 MHz)
12     a 125 kbps..."));
12 if (CAN0.begin(MCP_ANY, CAN_125KBPS, MCP_8MHZ)
13     == CAN_OK) {
13     Serial.println(F("MCP2515 inicializado
14         correctamente."));
14     CAN0.setMode(MCP_NORMAL);
15 } else {
16     Serial.println(F("Error al iniciar
17         MCP2515."));
17     while (1);
18 }
19 }
20
21 void loop() {
22 byte data[] = { 'H','e','l','l','o' };
23 byte len = 5;
24 if (CAN0.sendMsgBuf(0x123, 0, len, data) ==
25     CAN_OK)
25     Serial.println(F("Trama enviada -> ID=0x123,
26         Data=Hello"));
26 delay(500);
27 }
```

Listing 3: Código Arduino para módulo MCP2515.