Trabajo Fin de Grado

CONTROL DE INSTRUMENTACIÓN PARA MEDIDAS FÍSICAS MEDIANTE PROTOCOLO INALÁMBRICO DE BAJO COSTE

Daniel Enériz Orta

Directores:

Nicolás Medrano Marqués

Belén Calvo López

Universidad de Zaragoza | Facultad de Ciencias

Dpto. de Ingeniería Electrónica y Comunicaciones

Me gustaría dedicar las primeras líneas a agradecer a las personas que han echado una mano durante este TFG.

A Nicolás y Belén por ayudar con los problemas en del desarrollo y con la redacción de la memoria y a Óscar por su colaboración con los montajes.

Índice general

Resumen 1

1. Introducción 2

1.1 Motivación del trabajo 2

1.2 Objetivos 2

2. Recursos utilizados 4

2.1 Protocolo para control de instrumentos por computador 4

2.2 Protocolo y tecnología de comunicación 4

2.3 Soporte físico (hardware) 6

2.4 Soporte lógico (software): Python 7

3. Sistema de control y medida 12

3.1 Configuración del PC cliente 12

3.2 Configuración de los módulos locales – servidores Raspberry 14

4. Aplicaciones del sistema de control inalámbrico 17

4.1 Monitorización en paralelo de procesos independientes 17

4.2 Proceso de medida en un sistema disperso 19

4.3 Aplicación del módulo inalámbrico de control con alimentación por batería 21

5. Conclusiones y líneas futuras 25

Bibliografía 26

Anexos 28

A1. Programa *bash* para la instalación de paquetes de Python 28

A2. Librería `instin.py` 28

A3. Programa `autorun.py` 35

A4. Programa para la caracterización de dos filtros pasabanda `puesto_1.py` 39

A5. Programa para la medida del retraso de una señal y la atenuación en un cable coaxial (`coaxial.py`) 41

A6. Programa para la medida del voltaje de la batería que alimenta a la terminal `consumo.py` 45

Lista de Figuras 46

Lista de acrónimos 47

Resumen

La automatización de medidas en un proceso experimental facilita la adquisición de datos de forma sistemática a través del control de los instrumentos de medida por un *host*, permitiendo que los criterios de toma de datos y la configuración de los instrumentos puedan ser modificables atendiendo a las especificaciones indicadas por el usuario.

Todos los protocolos para comunicación y control de instrumentos en sistemas de medida automatizada –tanto los específicos de instrumentación como GPIB, VXI o PXI; o los estándares genéricos como TCP/IP o USB, entre los más extendidos– requieren de conexión por cable para la transmisión de información e instrucciones entre los diversos elementos del sistema de medida físico, lo que limita su ubicación. Para evitar esta restricción, algunos instrumentos de medida disponen de conectividad inalámbrica, fundamentalmente WiFi, permitiendo mayor flexibilidad en la selección de su ubicación. Sin embargo, esta solución no está muy extendida, pudiendo encontrarse sobre todo en sistemas específicos, como tarjetas de adquisición de datos y dataloggers.

Este trabajo desarrolla y valida un sistema de control inalámbrico para instrumentos de medida que dispongan de conexión USB y sean compatibles con el estándar VISA. Se basa en un ordenador de placa simple (*single board computer*, SBC) modelo Raspberry Pi Zero W con módulo de conexión WiFi de muy bajo coste y reducido factor de forma (65×30×5 mm^3). Esta plataforma se conecta al puerto USB del instrumento y se configura de modo que actúe de pasarela para los comandos y solicitudes de medida procedentes del PC que actúa de host del sistema de medida. La comunicación entre el PC de control y los SBC conectados en el puerto USB de los instrumentos se hace a través de una red local WiFi generada por un router comercial (Figura 1). El sistema propuesto permite controlar tantos SBC como sean necesarios y sincronizarlos para realizar medidas simultáneas, permitiendo desplegar instrumentos en grandes áreas.

La programación del sistema local (cada uno de los SBC conectados) se lleva a cabo sobre el sistema operativo Raspbian, una versión de Linux (Debian) adaptado a los módulos Raspberry. Los procesos de conexión a red WiFi, conexión VISA y recepción y envío de comandos e instrucciones entre el instrumento y el host se lleva a cabo a través de Python, siendo transparente para el usuario, que puede emplear la herramienta software que desee (Python, Matlab, LabView), simplemente utilizando la librería específica desarrollada para el control inalámbrico de instrumentación.

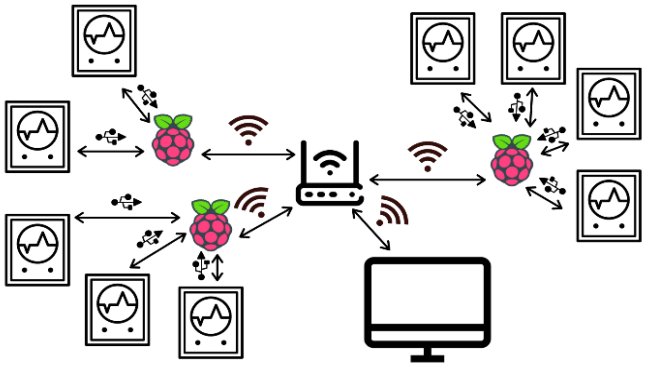

Figura 1: Esquema del sistema desarrollado

1. Introducción

1.1 Motivación del trabajo

La motivación principal del presente trabajo es la implementación de un sistema de bajo coste y reducido factor de forma que permita el control automático de instrumentación de forma inalámbrica. Puede haber muchas razones para justificarlo, pero vamos a resaltar las dos principales.

Por una parte, están las dimensiones del sistema monitorizado. Sistemas de gran tamaño pueden requerir cables de gran longitud para su conexión, lo que implica coste económico. La longitud máxima de los cables de datos de una instalación viene además limitada por las pérdidas asociadas, que pueden ser significativas a partir de unos pocos metros en el caso de estándares como el *Universal Serial Bus* (USB) o el *General Purpose Instrumentation Bus* (GPIB). Además, el tener cables dispuestos a lo largo de toda el área del sistema puede afectar a las condiciones de trabajo y dificulta la movilidad de la instrumentación.

Por otra parte, existe el riesgo de interferencia asociado al ruido electromagnético. Salvo estándares físicos diseñados específicamente, como el GPIB, una mayor longitud de cable de comunicaciones implica mayor riesgo de ser afectado por ruido externo. Es cierto que las comunicaciones son generalmente digitales, y que estas no se ven tan perjudicadas por el ruido como las analógicas. Pero la combinación de interferencia y pérdida de nivel de la señal puede dar lugar a errores en la transferencia de datos y el correspondiente mal funcionamiento del sistema de medida.

Además del soporte físico del proceso de medida, debe tenerse en cuenta el entorno de gestión y programación escogido para su consecución. El empleo de software libre tiene una gran ventaja: la accesibilidad. Hasta ahora, en toda nuestra experiencia durante el grado hemos controlado instrumentación mediante Matlab, herramienta de programación de pago. Esto implicaba una limitación, dado que fuera de los ordenadores de la Facultad no se tenía acceso al entorno de programación y no éramos capaces de poner en práctica nuestros conocimientos sobre el control de instrumentación.

Así, con la motivación de buscar una alternativa, decidimos usar software libre con la intención de hacer más accesible nuestro prototipo para su posible uso en el futuro. Además, con ello se obtenía una ventaja adicional al reducir el coste de nuestro sistema al evitar pagar por las posibles licencias necesarias.

1.2 Objetivos

El objetivo principal de este trabajo es desarrollar una plataforma hardware-software basada en un SBC de bajo coste que permita controlar la instrumentación necesaria para monitorizar un proceso de medida basado en instrumentos de medida comerciales mediante conexión inalámbrica estándar. Específicamente, las características que se pretenden conseguir en el desarrollo de la plataforma son:

(i) Sistema integral y genérico

Pretendemos que nuestra solución sea completamente funcional, autónoma y que no implique un consumo extra de recursos. Es decir, que el uso de la solución inalámbrica

propuesta sea transparente al usuario, pudiendo gestionar el sistema de medida de forma idéntica al empleo de una conexión directa por cable.

Por otro lado, no queremos restringir nuestra solución a un tipo de instrumentos o a un determinado fabricante. Nuestra intención es que sea lo más genérica posible, permitiendo controlar cualquier tipo de instrumento que tenga puerto USB.

(ii) Escalabilidad

Dado que lo que nos ha motivado a llevar a cabo este trabajo es la necesidad de cubrir sistemas de gran tamaño, es lógico que nos propongamos que el sistema no cuente con una restricción severa en cuanto al número de instrumentos que puedan incorporarse al sistema de medida ya que, en general, en sistemas más grandes van a ser necesarios más instrumentos.

(iii) Bajo coste

En la actualidad sólo se comercializan sistemas *datalogger* con conectividad WiFi para control inalámbrico de sistemas de medida multisensor. El objetivo de este trabajo es extender este tipo de conectividad a cualquier tipo de instrumentación compatible con conexión USB con una solución flexible con el menor coste posible.

Para ello, la presente memoria se estructura en cinco capítulos. El primero explica la motivación y los objetivos. El segundo capítulo presenta los diferentes recursos hardware y software que vamos a utilizar para llevar a cabo nuestro trabajo, el cual se describe en detalle en el tercer capítulo. En el capítulo cuarto validamos el sistema desarrollado, aplicándolo en tres escenarios diferentes. Finalmente, el capítulo quinto incluye las conclusiones y las posibles líneas futuras para poder ampliarlo.

Adicionalmente, los anexos 1 a 6 recogen información complementaria, como son la librería y el programa desarrollados para el control de la instrumentación vía WiFi y otros programas de control y automatización de medidas.

2. Recursos utilizados

En este capítulo vamos a presentar los recursos empleados en este trabajo de fin de grado. Atendiendo a sus características, podemos clasificarlos en: protocolo para control de instrumentos, protocolo y tecnología de comunicación, hardware y software.

2.1 Protocolo para control de instrumentos por computador

El estándar VISA (*Virtual Instrument Software Architecture*) es el protocolo de comunicación más generalizado que existe para el control de instrumentación por computador, empleado por grandes fabricantes del sector como por ejemplo *Keysight Technologies, National Instruments, Tektronix* o *Keithley*. VISA sirve de puente entre los programas de control del proceso de medida que se ejecutan en el computador, realizados en lenguajes de alto nivel como Matlab, Python o LabView y los propios instrumentos, adaptando el formato lógico-digital de la transmisión al estándar de conexión computador-instrumento empleado.

El hecho de que esté tan extendido se debe en gran parte a que incluye soporte de gran variedad de interfaces físicos de comunicación: desde dedicados a instrumentación, como GPIB, VXI o PXI, hasta los de propósito general como USB o TCP/IP[1].

Por su parte, existe otro protocolo estandarizado que define cómo debe ser la sintaxis de las instrucciones que deben interpretar los instrumentos, así como un pequeño subconjunto básico de comandos. Éste es el SCPI (*Standard Commands for Programmable Instruments*), incluido en el IEEE 488.2-1987 [1], el conocido estándar de códigos, formatos, protocolos y comandos para la comunicación a través del interfaz IEEE 488.1-1987 más comúnmente conocido como GPIB. Tras la aparición de otros buses de comunicación más generalizados como el USB, se adaptó para poder soportarlos.

Así pues, el SCPI permite el control de cualquier tipo de instrumento, sea cual sea su fabricante, con tal que los comandos asociados cumplan el estándar. Estos comandos son cadenas de texto codificado bajo el *American Standard Code for Information Interchange* (ASCII). SCPI también especifica el estándar del formato de los datos devueltos por la instrumentación ante una solicitud de medida, permitiendo su codificación tanto en binario (más compacto) como en ASCII (más fácil de interpretar).

A modo de resumen, podemos decir que VISA se encarga de la gestión y comunicación de instrumentos y SCPI es el encargado de decirnos cómo es la sintaxis y los comandos que usamos para ello.

2.2 Protocolo y tecnología de comunicación

Protocolo de comunicación TCP

El Protocolo de Control de Transmisión, o TCP por sus siglas en inglés, es un sistema de comunicaciones entre ordenadores conectados a una misma red que permite crear conexiones entre un servidor y un cliente para el intercambio de información. Asegura que los datos se entregan sin errores, debido a la inclusión del acuse de recibo (ACK), y en el

[1] Cabe comentar que aquí nos referimos a la comunicación que se establece mediante cables de red tipo ethernet, para la cual se usa también el protocolo TCP, pero de una forma alámbrica.

mismo orden en el que se transmitieron. La sencillez y la seguridad de este protocolo han provocado que sirva de base para otros protocolos más complejos como el HTTP (web), FTP (transferencia de ficheros), SMTP (correo electrónico), SSH (control remoto por la línea de comandos), entre otros.

En nuestro caso, el uso de TCP garantiza el envío correcto de datos entre el PC que controla la instrumentación y el módulo local de control que, conectado físicamente al instrumento, gestionará la comunicación inalámbrica. Para la comunicación necesitamos, en primer lugar, definir cuál de los dos dispositivos es el servidor (el que crea la línea de comunicación) y cuál el cliente (el que se conecta a ésta). Puesto que nos interesa tener más de una terminal que pueda controlar instrumentos en diferentes puntos, confiriendo escalabilidad a la solución propuesta, vamos a hacer que los módulos locales de control actúen como servidores, permitiendo así que el PC del usuario (que será el cliente) se pueda conectar a la terminal que considere oportuna en cada momento. Por poner un ejemplo ilustrativo, esta configuración es similar a una centralita telefónica de los años ochenta, donde el operador conecta la clavija en la conexión solicitada, quedando el resto del clavijero disponible para futuras comunicaciones.

Para establecer una comunicación entre servidor y cliente necesitaremos la dirección IP (*Internet Protocol*) del servidor y un puerto por el cual establecer la conexión. Tal y como hemos comentado, el TCP es de uso muy extendido, por lo que hay una serie de puertos que están reservados para aplicaciones específicas que trabajen sobre él (por ejemplo, el puerto para el HTTP, el protocolo de los navegadores web, es el 80). Existe una institución, la IANA (*Internet Assigned Numbers Authority*) que se encarga de definir qué puertos están libres y cuáles no. Para evitar conflictos con otros servicios que usan TCP, debemos usar un puerto superior o igual al 49152, ya que por debajo puede ser que el puerto esté en uso y la conexión no se podrá establecer. La gestión de comunicaciones la llevaremos a cabo mediante las funciones de la biblioteca `socket` de Python, que se comentará más adelante.

Finalmente, hablar de una limitación que debemos tener en cuenta a la hora de trabajar con el protocolo TCP. Se trata del tamaño máximo de segmento (*maximum segment size*, MSS), es decir, la cantidad de bytes máxima que podemos enviar en un paquete TCP. Este número lo configura el router y en nuestro caso es de 1452 bytes [2] [3].

Tecnología de comunicación WiFi

Para comunicar el PC con los módulos de gestión que se conectarán en los instrumentos vamos a usar WiFi como tecnología inalámbrica. Esta tecnología está muy extendida y permite la conexión de dispositivos electrónicos mediante el protocolo IEEE 802.11 a través de la banda de 2.4 GHz.

Para crear la red WiFi haremos uso de un router comercial, que se encargará de asignar una dirección IP a cada uno de los dispositivos que se conecten a ella. Serán estos valores de IP los que se emplearán para identificar a qué terminal se le envían las instrucciones. El área en la que se puede desplegar el sistema de medida vendrá determinada por el alcance de la red inalámbrica, ya que si nos encontramos fuera de este rango los módulos de gestión conectados a la instrumentación perderán la conexión inalámbrica, y el control

de los instrumentos. En caso de necesitar extender el rango, podemos hacerlo empleando dos métodos:

- Empleando un repetidor comercial. Estos repetidores tienen un coste muy bajo y están muy extendidos.
- Mediante una red VPN (*Virtual Private Network*) o red privada virtual. Podemos simular una red local, como la que estamos generando con nuestro router WiFi, si conectamos nuestra red a internet a través de una red pública. Esto implica que podríamos controlar nuestras terminales de medida desde cualquier distancia, siempre y cuando estas cuenten con conexión a internet y el PC del usuario también. Para ello existen varios servicios para la gestión de VPNs, como son Hamachi u OpenVPN [4] [5].

De cualquier manera, creemos que la posibilidad de aumentar el área del sistema es una función que dota a nuestro sistema de mayor versatilidad, pero no la vamos a poner en práctica al quedar fuera de los objetivos del presente trabajo.

2.3 Soporte físico (hardware)

Como se ha indicado en la introducción, el control local del instrumento se llevará a cabo empleando recursos hardware que permitan gestionar a través de la tecnología WiFi el intercambio de datos e instrucciones con el PC de control que hace de *host*, así como con el (o los) instrumento(s) conectados, y que disponga de potencia computacional suficiente para ejecutar funciones de alto nivel, todo ello en un dispositivo de reducidas dimensiones, bajo coste y consumo. Las placas Raspberry son ejemplos típicos de lo que se conoce como ordenadores de placa simple con estas características. Tienen su origen en Reino Unido en torno a 2006 con fines educativos. De los diferentes modelos disponibles, el seleccionado para este trabajo, una Raspberry Pi Zero W, dispone de procesador mononúcleo Broadcom BCM2835 de 1GHz, 512 MB de RAM, y slot para introducir una tarjeta microSD, con unas dimensiones totales de 65mm×30mm×5mm. En cuanto a conectividad inalámbrica, el dispositivo seleccionado consta de antena trapezoidal integrada en la propia PCB que permite conexión Bluetooth 4.1 de baja energía y WiFi (2.4GHz 802.11n). Asimismo, tiene un puerto mini-HDMI para pantalla, un micro-USB para alimentación y otro para uso como USB típico.

Para permitir que cada uno de los módulos locales pueda controlar varios instrumentos, cada Raspberry lleva conectado un hub USB Zero4U de UUGear [6]. Este módulo permite

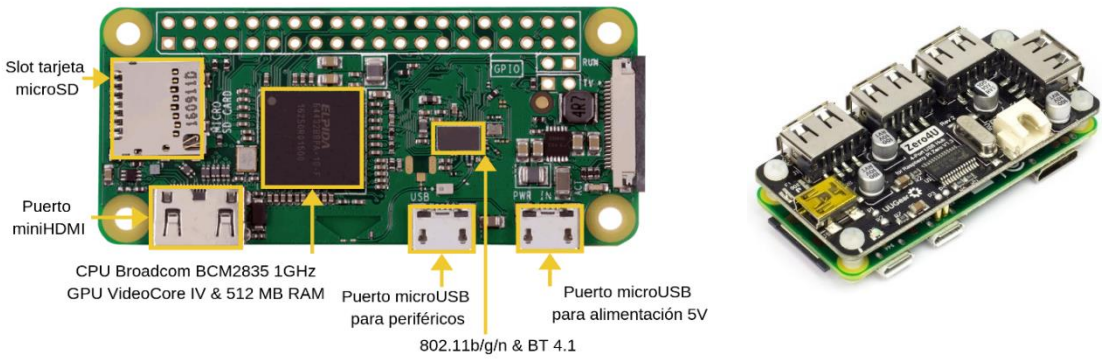

Figura 2: (izda.) Esquema del hardware de la Raspberry Pi Zero W; (dcha.) Vista del módulo Zero4U acoplado a una Raspberry Pi Zero W

añadir cuatro puertos USB de tipo A para conexión a periféricos (downstream) más un miniUSB para posibilitar la conexión a un PC (upstream). Este hub USB se comunica directamente con la placa de la Raspberry mediante cuatro pines de tipo pogo [7].

La gestión de los recursos de la Raspberry se lleva a cabo a través de un sistema operativo específico para estas placas llamado Raspbian. Consiste en una versión del sistema operativo libre Debian, basado a su vez en el núcleo de Linux. La distribución para estos módulos hardware incluye la versión 3.5 de Python, que será el que emplearemos para programar la operación local de los módulos y su interacción con los instrumentos y otros periféricos, como se verá más adelante.

2.4 Soporte lógico (software): Python

Python es uno de los lenguajes de programación interpretado multiplataforma más conocidos debido a su sintaxis y a su versatilidad. A continuación, comentamos los puntos que nos han llevado a elegir a Python como lenguaje a utilizar para el desarrollo de nuestro sistema:

- Es un lenguaje de código abierto. El uso de Python está en línea con el objetivo de utilizar software libre, evitando así tener que utilizar programas de pago como Matlab para el control de instrumentación.
- Es multiplataforma, lo que permite desarrollar código en diferentes soportes físicos. Además, permite optimizar el uso de los recursos hardware, lo que hace innecesario el empleo de plataformas potentes, lo que permite utilizar el módulo Raspberry Pi Zero W propuesto.
- Está ampliamente extendido, existiendo un gran número de desarrolladores de aplicaciones y librerías, e incluye varias bibliotecas para la gestión de comunicaciones TCP, control de instrumentación mediante VISA e incluso tarjetas de adquisición de datos.
- Finalmente, dado que Python 3.5 está incluido en el sistema operativo Raspbian no era necesario una instalación específica de este lenguaje.

Así pues, podemos crear un programa de Python que se ejecutará en los terminales Rasberry y que se encargará de crear un servidor TCP que quedará a la escucha de órdenes del cliente, el PC. Dichas órdenes serán instancias SCPI que se llevarán a cabo en los instrumentos conectados, permitiendo así el control remoto de éstos.

De forma similar a otras plataformas de programación, Python reúne en librerías o bibliotecas las funciones desarrolladas para un tipo concreto de aplicación (representación gráfica, cálculo científico, control de instrumentación). Para entrar en más profundidad en el uso de Python, presentaremos aquellas que vamos a usar en nuestro trabajo y describiremos cómo se integran en nuestro sistema.

- Librería `pyvisa.py`. El paquete `pyvisa` está diseñado para adaptar el protocolo VISA a Python, permitiendo así el control de instrumentación. En la actualidad los únicos lenguajes para los cuales existen librerías oficiales de VISA son LabView, C y Visual Basic. Alternativamente Matlab desarrolló su propia librería para el control de instrumentación y, para Python, la comunidad ha desarrollado `pyvisa`. De esta forma, `pyvisa` nos va a permitir realizar las operaciones de conexión y comunicación con los instrumentos a través del estándar VISA indicado anteriormente. Veamos un ejemplo de cómo se usa:

```
import visa #Importamos la librería
rm = visa.ResourceManager() #Abrimos el gestor de objetos
osc = rm.open_resource('USB0::0x0957::0x179B::MY50512084::INSTR') #Abrimos un osciloscopio
osc.write('aut') #Le decimos que autoajuste la representación en pantalla
```

- Librería pyvisa-py.py. Una de las posibilidades para conectar instrumentación a un computador es emplear NI-VISA, el *backend* de National Instruments. Básicamente, su misión es la equivalente a la que desempeña un driver en la gestión de un recurso hardware. Sin embargo, National Instruments actualmente solo da soporte en los sistemas operativos de Microsoft, Apple y algunos de Linux. No existe por tanto una versión para Debian, si bien se ha desarrollado `pyvisa-py`, que se define como un *backend* de `pyvisa`, es decir, una librería de soporte para la comunicación con los buses compatible con Python en cualquier sistema operativo, además de código abierto. Esto no cambia nada del código anterior, excepto que para decirle a `pyvisa` que queremos usar `pyvisa-py` deberemos indicárselo en el argumento en el gestor de objetos:

```
rm = visa.ResourceManager('@py') #Queremos usar pyvisa-py
```

- Librería socket.py. Si `pyvisa` y `pyvisa-py` son los paquetes que nos van a servir para el intercambio de datos con los instrumentos, el paquete `socket` es el que vamos a usar para la comunicación entre Raspberry y PC. Es el módulo de Python más extendido para el uso del protocolo TCP por su sencillez. Vamos a ver un ejemplo simple en la

CÓDIGO SERVIDOR (RASPBERRY)

```
import socket

#Elegimos el puerto que vamos a usar:
PORT = 50001


#Creamos el objeto socket:
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1) #Nos permite usar
el mismo puerto
try:
    s.bind(('', PORT)) #Creamos el servidor
except socket.error as err:
    print('Error al crear el servidor:')
    print(err)
#Limitamos el número de clientes que se
pueden conectar al servidor:
s.listen(1)


#Nos conectamos al cliente que tenga
nuestro puerto:
(conn, addr) = s.accept()
#Preparamos para recibir el mensaje
```

CÓDIGO CLIENTE (PC)

```
import socket
#Datos del servidor al que nos conectamos
HOST = '192.168.1.2'
PORT = 50001
#Creamos el objeto socket:
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1) #Nos permite usar
el mismo puerto









#Nos conectamos al servidor
s.connect((HOST, PORT))
#Envío de datos:
msg = 'Hola!'
```

```
BUFF = 512 #Ajustamos el tamaño del buffer de datos


data = conn.recv(BUFF) #Se recibe en tipo byte

msg = data.decode('ASCII') #Pasamos a tipo string en formato ASCII

#Envío de datos

msg = 'Hemos recibido ' + msg

data = msg.encode('ASCII') #Conversión a tipo byte

conn.send(data) #Lo enviamos al cliente


#Cerramos comunicaciones

conn.close()

s.close()
```

```
data = msg.encode('ASCII') #Conversión a tipo byte

s.send(data) #Envío



#Recepción de datos

BUFF = 512 #Ajustamos el tamaño del buffer de datos


data = s.recv(BUFF) #Recibiendo datos del servidor

msg = data.decode('ASCII')

 #Cierre de la conexión

s.close()
```

Figura 3, donde la columna de la izquierda corresponde al código que se ejecuta en el módulo Raspberry y la de la derecha el código del cliente, el PC.

```
CÓDIGO SERVIDOR (RASPBERRY)

import socket

#Elegimos el puerto que vamos a usar:

PORT = 50001


#Creamos el objeto socket:

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo puerto

try:

    s.bind(('', PORT)) #Creamos el servidor

except socket.error as err:

    print('Error al crear el servidor:')

    print(err)

#Limitamos el número de clientes que se pueden conectar al servidor:

s.listen(1)


#Nos conectamos al cliente que tenga nuestro puerto:

(conn, addr) = s.accept()

#Preparamos para recibir el mensaje

BUFF = 512 #Ajustamos el tamaño del buffer de datos
```

```
CÓDIGO CLIENTE (PC)

import socket

#Datos del servidor al que nos conectamos

HOST = '192.168.1.2'

PORT = 50001

#Creamos el objeto socket:

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo puerto



#Nos conectamos al servidor

s.connect((HOST, PORT))

#Envío de datos:

msg = 'Hola!'

data = msg.encode('ASCII') #Conversión a tipo byte
```

```
data = conn.recv(BUFF) #Se recibe en tipo byte
msg = data.decode('ASCII') #Pasamos a tipo string en formato ASCII
#Envío de datos
msg = 'Hemos recibido ' + msg
data = msg.encode('ASCII') #Conversión a tipo byte
conn.send(data) #Lo enviamos al cliente

#Cerramos comunicaciones
conn.close()
s.close()
```

```
s.send(data) #Envío


#Recepción de datos
BUFF = 512 #Ajustamos el tamaño del buffer de datos

data = s.recv(BUFF) #Recibiendo datos del servidor
msg = data.decode('ASCII')
 #Cierre de la conexión
s.close()
```

Figura 3: Ejemplo de una conexión TCP entre cliente (izquierda) y servidor (derecha) usando la librería `socket` de Python. El servidor envía un dato al cliente y este se lo devuelve.

Por otra parte, como vemos, es muy simple entablar comunicación entre dos IPs usando la librería `socket` de Python. Es más, podemos ir más allá y establecer una conexión TCP entre un programa en Python y cualquier otro lenguaje que tenga una librería que permita el uso de este protocolo, como pueden ser LabView, C o Matlab. Presentamos en la

```
PROGRAMA DEL SERVIDOR (PYTHON)
import socket
PORT = 50007
#Creamos el objeto socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#Creamos el servidor
s.bind(('', PORT))
s.listen(1)
#Creamos la conexión con el cliente
(conn, addr) = s.accept()
#Recepción del mensaje
data = conn.recv(512)
mesg = data.decode('ASCII'))
#Cierre de conexiones
conn.close()
s.close()
```

```
PROGRAMA DEL CLIENTE (MATLAB)



% Nos conectamos al servidor
t = tcpip('192.168.1.2', 50001);
fopen(t);
% Mandamos un mensaje
fwrite(t, 'Hola!');


% Cierre de conexión
fclose(t);
```

Figura 4 un ejemplo muy sencillo en el que un programa Python crea un servidor TCP y un cliente se conecta a él desde Matlab.

```
PROGRAMA DEL SERVIDOR (PYTHON)
import socket
PORT = 50007
#Creamos el objeto socket
```

```
PROGRAMA DEL CLIENTE (MATLAB)
```

```
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
#Creamos el servidor
s.bind(('', PORT))
s.listen(1)
#Creamos la conexión con el cliente
(conn, addr) = s.accept()
#Recepción del mensaje
data = conn.recv(512)
mesg = data.decode('ASCII'))
#Cierre de conexiones
conn.close()
s.close()
```

```
% Nos conectamos al servidor
t = tcpip('192.168.1.2', 50001);
fopen(t);
% Mandamos un mensaje
fwrite(t, 'Hola!');

% Cierre de conexión
fclose(t);
```

Figura 4: Ejemplo de una conexión TCP entre un servidor (izquierda) establecido con la librería `socket` de Python y un cliente usando Matlab (derecha).

3. Sistema de control y medida

Una vez presentados los elementos hardware y software constitutivos del sistema de gestión y control de medida inalámbrico procedemos a describir su implementación. Ésta requiere en esencia dos pasos: instalación en el PC cliente de las librerías necesarias para enviar instrucciones e interrogar a los módulos locales a los que se conecte para obtener las medidas; e instalación del software complementario en los módulos locales Raspberry.

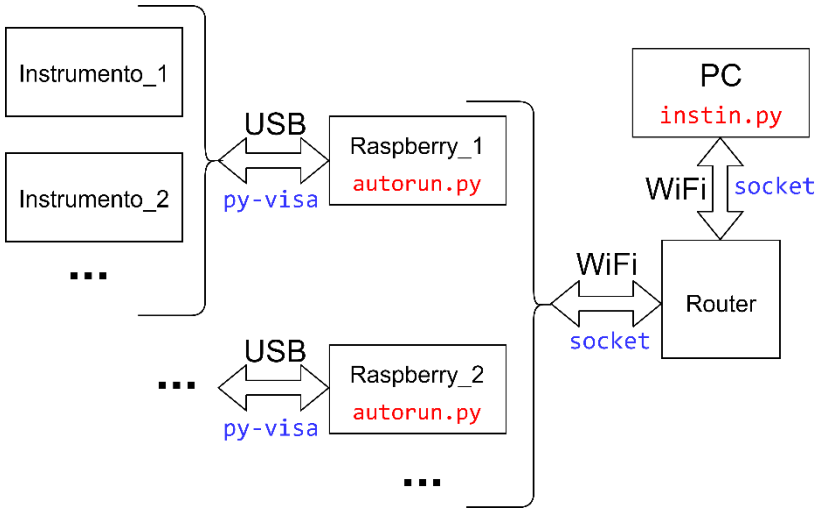

Figura 5: Esquema de las conexiones hardware para la comunicación inalámbrica que incluye las librerías utilizadas (azul) y los códigos desarrollados (rojo).

3.1 Configuración del PC cliente

La operación del PC como cliente se ha llevado a cabo desarrollando en Python las funciones necesarias para la gestión de un proceso de medida basado en los módulos Raspberry. Estas funciones, integradas en la librería `instin.py`, desarrollada para este trabajo, se complementan con las correspondientes funciones desarrolladas en el módulo local, alojadas en el programa `autorun.py`, capaces de interpretar los mensajes enviados por el cliente. Las funciones de la librería empleada en el PC se describen a continuación.

Librería `instin.py`

La comunicación entre el PC que controla el proceso de medida y los módulos locales conectados a los diferentes instrumentos se realiza mediante el envío de mensajes compuestos por dos partes: una cabecera de un carácter ASCII de longitud y un argumento (Figura 6). Dependiendo del valor de la cabecera, el módulo Raspberry determina qué hacer con el argumento del mensaje. Las funciones de la librería `instin.py` se encargarán de generar los mensajes que se envían a cada módulo local en función de sus argumentos. La transmisión y recepción de datos a través de TCP se encuentra integrada en el código de las funciones de la librería, siendo transparente para el usuario. Puede consultarse la librería completa en el Anexo A2.

Cabecera (un carácter)	Resto del mensaje

Figura 6: Formato de los mensajes que recibe `autorun.py`

Función *sincro(IP_addr, port_num, hora)*

Es la función que debe ejecutarse cuando se va a iniciar un proceso de medida al conectarse a uno de los módulos locales por primera vez. Si éstos se apagan, sus relojes pierden la hora, por lo que cada vez que nos conectemos debemos mandar la hora del PC del usuario para su sincronización. La función establece la conexión con la Raspberry cuya IP se indica y le envía la hora. Una vez hecho esto, cierra la conexión. Tiene como entradas la IP del módulo local, el puerto de conexión y la hora del PC en formato `'%m%d%H%M.%S'`. Por las características de esta función (es la primera en ejecutarse para establecer una conexión), la cabecera del mensaje enviado no tiene uso.

Función *instr_ID = open_inst(IP_addr, port_num, VISA_addr)*

Esta función se encarga de abrir una conexión a través del puerto `port_num` al módulo local cuya dirección es `IP_addr` y envía un mensaje cuyo argumento es la dirección VISA (`VISA_addr`) de uno de los instrumentos conectados a la Raspberry precedida por un "0" como cabecera. La función devuelve `instr_ID`, el identificador que el módulo local le asigna al instrumento para su posterior identificación en el proceso de medida. Tras finalizar la comunicación, la función cierra la conexión.

Función *write(IP_addr, port_num, instr_ID, SCPI_comm)*

Transmite a través del puerto `port_num` el comando SCPI `SCPI_comm` al instrumento con identificador local `instr_ID` al módulo Raspberry identificado por la dirección `IP_addr`. La cabecera debe ser un número diferente de "0". Una vez enviado este paquete, la conexión se cierra.

Función *data = query(IP_addr, port_num, instr_ID, SCPI_comm)*

Es similar a la función `write(.)`, sólo que tras el envío del identificador y el comando la función queda a la espera de recibir la respuesta del instrumento. Seguidamente cierra la conexión y devuelve la respuesta al programa principal.

Función *close_term(IP_addr, port_num)*

Transmite a través del puerto de comunicación `port_num` a la Raspberry identificada con la dirección WiFi `IP_addr` la instrucción de cerrar la comunicación y reiniciarse. El mensaje está compuesto por la palabra `close` precedida por un número diferente de "0" como cabecera. En cuanto el módulo local detecta que el mensaje contiene `close`, la Raspberry se reinicia y permanece a la espera de recibir un nuevo mensaje proveniente de la función `sincro(.)`.

Función *send_program(IP_addr, port_num, file_name)*

Las funciones descritas anteriormente permiten por parte del PC cliente el control instrucción a instrucción, de manera consecutiva en cada instrumento, del proceso de medida, de forma que se reproduce el proceso de un sistema de instrumentación cableado estándar. Sin embargo, la arquitectura de PC-cliente + servidores WiFi Raspberry propuesta en este TFG permite delegar a los módulos la gestión del proceso de medida local, de manera que cada uno de ellos sea responsable de una parte de las medidas de manera independiente. Es decir, delegar la gestión del control de la instrumentación a cada Raspberry, de forma que puedan lanzarse varios procesos de medida independientes cuyos resultados sean transmitidos al PC cliente una vez concluidos. Esta posibilidad la proporciona la función `send_program(.)`, que transmite a través del puerto `port_num` al

módulo local identificado por la dirección WiFi `IP_addr` el archivo que contiene los comandos que deben ejecutar los instrumentos conectados a esa Raspberry.

Tras abrir la conexión, la función envía un mensaje que contiene el nombre del fichero de comandos, incluido el directorio donde debe guardarse, precedido por el carácter `'p'` como cabecera (un ejemplo sería `p/home/pi/Desktop/ejecutable.py`) y queda a la espera de respuesta. Tras recibir el mensaje del módulo local, la función enviará a la Raspberry el número de archivos de salida que el programa debe generar seguido por sus nombres completos (incluyendo el directorio de almacenamiento en la Raspberry).

Tras un mensaje de confirmación por parte del módulo local, el PC cliente enviará el archivo que contiene el código que se ejecutará localmente. Teniendo en cuenta la limitación del tamaño de los paquetes asociada al MSS (Sección *Protocolo de comunicación TCP*), este archivo debe fraccionarse en paquetes que no superen 1452 bytes. En nuestro caso, dividimos los ficheros en paquetes cuyo tamaño sea múltiplo de 512 B, añadiendo el carácter *salto de línea* (`'\n'`) tantas veces como sea necesario en caso de ser de tamaño inferior. Una vez hecho esto, se comunica a la Raspberry el número de paquetes que componen el código ejecutable y a continuación se envían; localmente el módulo se encargará de encadenarlos de nuevo en un único archivo que iniciará su ejecución de forma automática.

Tras su ejecución, y una vez guardados localmente los archivos de salida, el PC cliente los recibirá desde el módulo local Raspberry usando el mismo método que al enviar el archivo del programa: rellenados hasta un múltiplo de 512 y recibiendo previamente el número de paquetes de 512 bytes que debe recibir. Cuando el último fichero es escrito en el PC de control se cierra la conexión.

3.2 Configuración de los módulos locales – servidores Raspberry

Cada una de nuestras terminales cuenta con una Raspberry Pi Zero, un hub USB Zero4U, un cable de alimentación con terminación en microUSB y una microSD. Es en esta tarjeta de memoria, previamente formateada, donde se introduce el sistema operativo que queremos instalar. En nuestro caso, como ya hemos comentado, elegimos el sistema operativo propio de Raspberry, Raspbian.

Una vez instalado el sistema operativo, se instalarán los paquetes de Python necesarios. Para hacer este proceso automático hemos creado un *script bash* que contiene todos los comandos precisos (Anexo A1). Una vez instalados los paquetes de Python en la Raspberry, la conectaremos a la red WiFi que queramos usar como puente entre ella y el PC cliente.

Archivo `autorun.py`

Cada vez que se inicie la Raspberry se ejecutará este programa, de forma que el módulo local queda a la espera de recibir las instrucciones procedentes del PC cliente (Anexo A3). Para conseguir este inicio automático, se configura el administrador de procesos incluido en Raspbian, `crontab`, para que se encargue de lanzar `autorun.py` cada vez que la Raspberry se inicie. La relación entre las funciones asociadas a la librería `instin.py` del PC cliente y las instrucciones que ejecuta el módulo servidor remoto pueden observarse en la Figura 7.

Al establecer la primera conexión, la Raspberry recibirá desde el PC la hora y fecha [procedente de la función *sincro(.)* del cliente], que empleará para poner al servidor local en hora con el PC. Seguidamente la Raspberry entra en un bucle infinito en el que la terminal queda a la espera de nuevas solicitudes de conexión desde el PC. Cada vez que se abra una conexión, la Raspberry queda a la espera de un mensaje con el formato indicado en la Figura 6. Dependiendo del carácter de la cabecera del mensaje se ejecutarán las instrucciones correspondientes.

Apertura de conexión con un instrumento [PC: función *open_inst(.)*]

Si el carácter de cabecera es un 0, llamaremos la función de la librería `pyvisa` para abrir la conexión con el instrumento cuya dirección VISA corresponde al resto del mensaje. Guardaremos el objeto instrumento generado al abrir la conexión en un vector que previamente habremos inicializado y, a continuación, el módulo local envía al PC la posición del instrumento en el vector de instrumentos a modo de identificador, cerrando a continuación la conexión a la espera de otra solicitud.

Instrucción o pregunta al instrumento [PC: funciones *write(.)* o *query(.)*]

Cuando el carácter de cabecera del mensaje recibido sea un número natural distinto de 0, el programa a priori interpreta la recepción de un comando o una pregunta al instrumento cuyo identificador local se indica en el mensaje. El resto del mensaje será el comando SCPI que queremos enviar. Si éste contiene un carácter `'?'` se corresponde con una pregunta, por lo que tras remitirlo al instrumento el módulo local quedará a la espera de respuesta por parte del instrumento, enviándola a continuación al PC. En otro caso, el comando es una instrucción, siendo sólo necesario mandarlo al instrumento. Tras cada una de estas dos operaciones la conexión se cierra y se vuelve a la primera línea del bucle infinito en la que la Raspberry queda a la espera de una nueva conexión.

Reinicio de la Raspberry [PC: función *close_term(.)*]

La última funcionalidad que nos queda por comentar es la de reiniciar la terminal. Para ello basta con enviar un número natural distinto de cero seguido de la palabra `close` a modo de mensaje. Una vez hecho esto la Raspberry se reinicia esperando una nueva sincronización.

Función local *launch_program(.)* [PC: función *send_program(.)*]

Para habilitar la ejecución local de programas de medida en una Raspberry de forma autónoma respecto al PC cliente, el módulo local implementa en el programa `autorun.py` la función *launch_program(.)*, que se ejecutará al recibir el comando correspondiente [enviado por el PC mediante la función *send_program(.)*]. Esta función se encarga de recibir el programa que se requiere ejecutar y devolver los archivos de salida que se generen. Cuando la función acaba, la conexión se cierra y la terminal vuelve a quedar a la espera de una conexión.

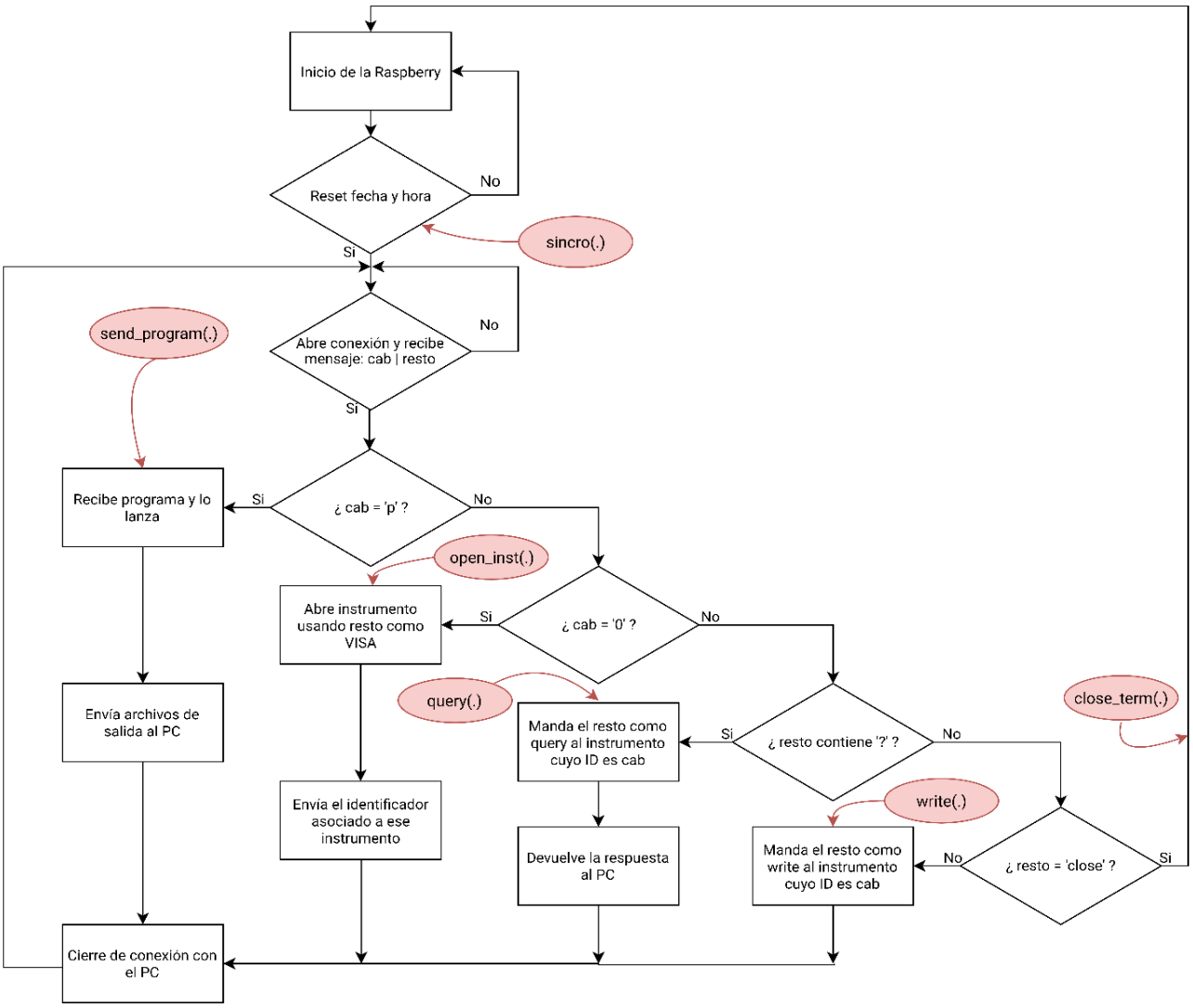

Figura 7: Diagrama de flujo de un módulo local y funciones del cliente implicadas (en rojo)

4. Aplicaciones del sistema de control inalámbrico

4.1 Monitorización en paralelo de procesos independientes

Como se ha indicado en el capítulo anterior, con la función *send_program(.)* es posible precargar un programa de control en cada uno de los módulos Raspberry, de forma que se ejecuten un conjunto de procesos de caracterización y medida de manera independiente. Para verificar esta funcionalidad, nos proponemos como ejemplo caracterizar en paralelo ocho filtros pasabanda (cuatro de frecuencia central de 2 kHz y otros cuatro de 20 kHz) empleando los osciloscopios con generador incorporado Agilent DSO X2002A controlados por los módulos locales a través de su entrada USB. Las cuatro Raspberry Pi Zero de que disponemos se configurarán de forma que cada una gestione las medidas de dos de los procesos localmente, conectando dos osciloscopios a cada una de ellas. Adicionalmente, usaremos una fuente DC para alimentar los sistemas. La Figura 8 dcha. muestra uno de los puestos de medida local.

Para la puesta en funcionamiento del sistema completo de medida, en primer lugar, se activa el router WiFi que creará la red local. A continuación, se conectan a la alimentación de forma consecutiva las cuatro terminales Raspberry que controlarán los instrumentos, asignándoles el router a cada una dirección IP conforme se conecten a la red WiFi. Para conocer sus valores, necesarios para las posteriores comunicaciones entre módulos, consultaremos desde un PC los diferentes dispositivos conectados y sus números de IP a través del servidor web que mantiene el router en la dirección `192.168.1.1`.

Con esta configuración cada módulo Raspberry controla dos procesos de medida simultáneos. Por ello, crearemos cuatro códigos Python que se ejecutarán cada uno en uno de los módulos locales y que controlarán dos osciloscopios, creando los diferentes archivos que almacenarán los datos de la caracterización de cada uno de los filtros. Estos programas los denotaremos como `puesto_i.py`. Podemos encontrar el archivo `puesto_1.py` en el Anexo A4.

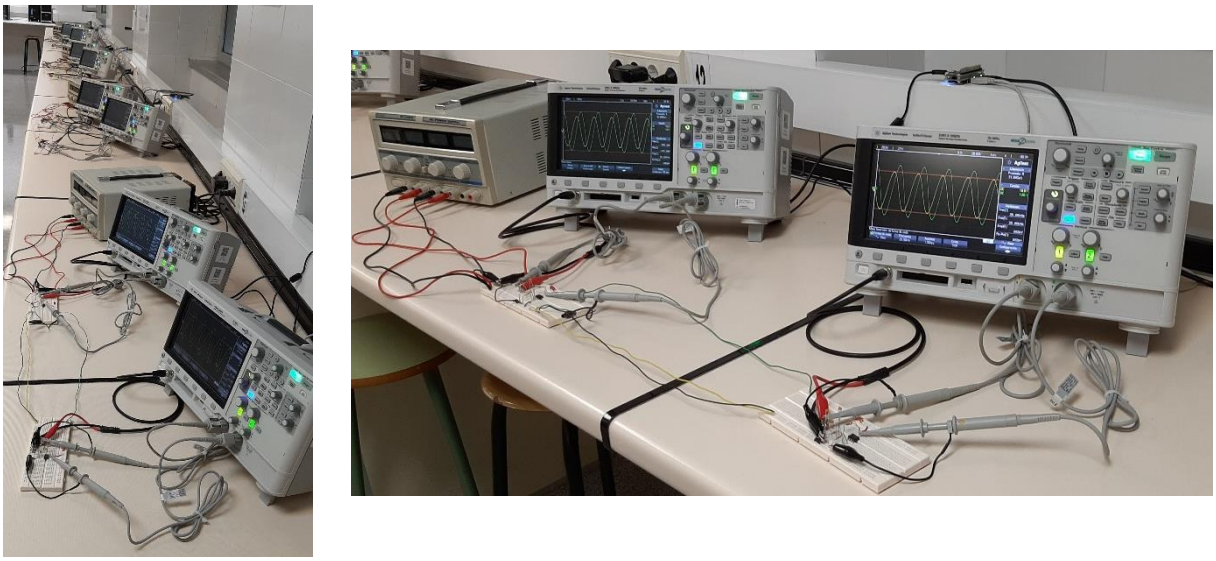

Figura 8: (izda.) Sistema de caracterización de 8 filtros en paralelo; (dcha.) detalle de uno de los puestos.

En el módulo central, el PC cliente ejecutará simultáneamente los archivos Python (`launcher_i.py`) que incluyen las funciones necesarias para la conexión sucesiva de las cuatro Raspberry [con la función *sincro(.)* para cada módulo local], del envío de los programas de ejecución, nombres de los archivos de salida y recepción de los resultados [función *send_program(.)*], para terminar con el cierre de los terminales [función *close_term (.)*], una vez acabado el proceso de medida.

Además, debemos tener en cuenta que al ejecutar en paralelo cuatro procesos con comunicación TCP será necesario incluir un archivo `autorun.py` con un puerto diferente en cada una de las Raspberry. Nosotros hemos optado por incluir en cada uno de los `autorun_i.py` la conexión TCP en los correspondientes puertos $5000i$ ($i = 1$ a 4).

A continuación, presentamos el programa `launcher_1.py` (Figura 9). El resto de los programas `launcher_i.py` son análogos a este, sólo que cambiando la IP (pondremos la de la Raspberry a la que nos queramos conectar), el número de puerto TCP y los nombres de los archivos de entrada y salida que están como argumentos en la función *send_program(.)* de nuestra librería.

```
import instin as i
import time


HOST = '192.168.1.5'
PORT = 50001
print('PC: ' + time.ctime(time.time()))
i.sincro(HOST, PORT, time.strftime('%m%d%H%M.%S'))


i.send_program(HOST, PORT, 'puesto_1.py', 'data_p1_osc0.txt', 'data_p1_osc1.txt')
print('Cerrando la terminal')
i.close_term(HOST, PORT)
```

Figura 9: Programa `launcher_1.py`.

Así pues, una vez lanzados los programas `launcher_i.py` estos se encargarán de mandar los programas `puesto_i.py` a las Raspberry Pi y éstas los ejecutarán, comenzando de manera independiente la toma de datos. Una vez acabado el proceso de medida, se generarán dos archivos de salida en cada módulo local, `data_pi_osc0.txt` y `data_pi_osc1.txt` (para cada uno de los dos osciloscopios a los que se conecta cada Raspberry), que contienen la ganancia y desfase salida-entrada del circuito en función de la frecuencia de la señal. Finalmente, las Raspberry enviarán todos los archivos de salida al PC.

El proceso de medida es bastante rápido, pero vemos cómo hay alguna diferencia en función del puesto en el que se lance el programa. Por ejemplo, para los puestos en los que los dos filtros tenían frecuencia central de 20 kHz, la toma de 100 puntos ha tardado 4,5 min. En cambio, para uno de los puestos de caracterización con los filtros de 2 kHz de frecuencia central el tiempo de medida ha sido de 2,25 min y en el otro casi 13 min. Creemos que el módulo Raspberry de este último puesto debía tener algún defecto de hardware y/o de software que ha limitado su velocidad de operación, ya que tanto en la

instalación como en las pruebas anteriores a las demostraciones vimos cómo tomaba más tiempo en realizar las mismas tareas que el resto de Raspberry.

A modo de conclusión, creemos que esta prueba es un buen ejemplo de cómo nuestro sistema puede ser útil en ambiente en el que se necesite caracterizar varios sistemas independientes de forma inalámbrica, sin necesidad de invertir un mayor tiempo que por el método tradicional: a través de un cable USB.

4.2 Proceso de medida en un sistema disperso

Si en la anterior sección veíamos cómo nuestro trabajo podría servir para controlar varios sistemas de medida trabajando de forma independiente, en este vamos a ver cómo podemos usarlo en un sistema extenso donde el empleo de instrumentación controlada de forma centralizada con un estándar por cable se hace difícil. Para ello, proponemos una experiencia académica: supongamos que se quiere determinar la distancia entre dos puntos de un cable coaxial de comunicaciones, al que tenemos acceso en ciertos puntos, uno de los cuales estableceremos como punto inicial. Para realizar la experiencia, disponemos en el laboratorio de un cable coaxial de 11 m de longitud que emula un cable de datos desplegado y un generador arbitrario de funciones Tektronix AFG3252 conectado a la entrada del cable, que proporciona pulsos de 1 MHz de frecuencia con un ciclo de trabajo del 1 %. Para observar el comportamiento del cable ante dicha señal, se dispone de dos osciloscopios Tektronix DPO4104 monitorizando la señal del cable a una distancia conocida desde el generador. En esas condiciones, no es posible estimar la velocidad de fase en el cable usando el tiempo transcurrido entre el paso de la señal por uno de los osciloscopios (situado, por ejemplo, junto al generador) hasta su llegada al segundo punto monitorizado, si están suficientemente alejados para no poder compartir una señal de disparo común[2]. Sin embargo, podemos medir la velocidad monitorizando las señales del cable en un único punto si usamos la señal reflejada en el final del cable. Podemos expresar el coeficiente de reflexión (Γ) de una señal desplazándose por un cable en función de la impedancia característica (Z_0) de éste y la impedancia de carga (Z_L) en la terminación [8]:

$$\Gamma = \frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} \quad (1)$$

De esta forma, en condiciones de circuito abierto ($Z_L = \infty$), el coeficiente de reflexión es $+1$ y, por tanto, la amplitud de la señal reflejada (V_r) es igual a la de la entrante (V_i). En cortocircuito ($Z_L = 0$) ocurre al contrario, el coeficiente de reflexión es -1 y la amplitud de la señal reflejada es $-V_i$. Así mismo, la señal que introduce el generador está ajustada para que el coeficiente de reflexión se anule cuando $Z_L = 50\ \Omega$.

Por ello, si configuramos el generador de ondas para enviar pulsos y dejamos el extremo del cable en circuito abierto, la señal que se verá en un punto de la línea de transmisión será como la mostrada en la Figura 10 dcha. En ella se puede observar un primer pulso,

[2] El osciloscopio DPO4104 no dispone de la opción de marcado de tiempo (*timestamp*) del disparo disponible en familias superiores como los DPO/DSA/MSO 5K 7K 70K, que permiten obtener el instante de tiempo en el que se produjo el disparo con una resolución del reloj de picosegundos, y que sí permitirían establecer el tiempo transcurrido entre ambas medidas comparando las lecturas de los dos osciloscopios y, con ello, obtener la velocidad de fase [16].

el inicial, seguido de otro cuya altura es menor (el pulso reflejado), debido a la atenuación. Si conocemos la distancia desde el punto monitorizado al de reflexión, es posible determinar la velocidad de propagación de la señal a partir del tiempo entre los dos pulsos[3]. En nuestro caso, se muestreó la señal a 7,5 m de distancia desde el final del cable. Para ello, desarrollamos un código que controla de forma inalámbrica el generador de ondas y el osciloscopio, separados 3,5 m. El programa configura las características de la señal que emite el generador, y determina el retraso entre las señales original y reflejada, así como la altura de los dos pulsos que se ven en pantalla del osciloscopio. En este caso, el PC centraliza la gestión de la generación de la señal y la medida del instrumento localizado en el recorrido del cable, obteniendo la información a partir de los datos recogidos. Incluimos dicho código en el Anexo A5.

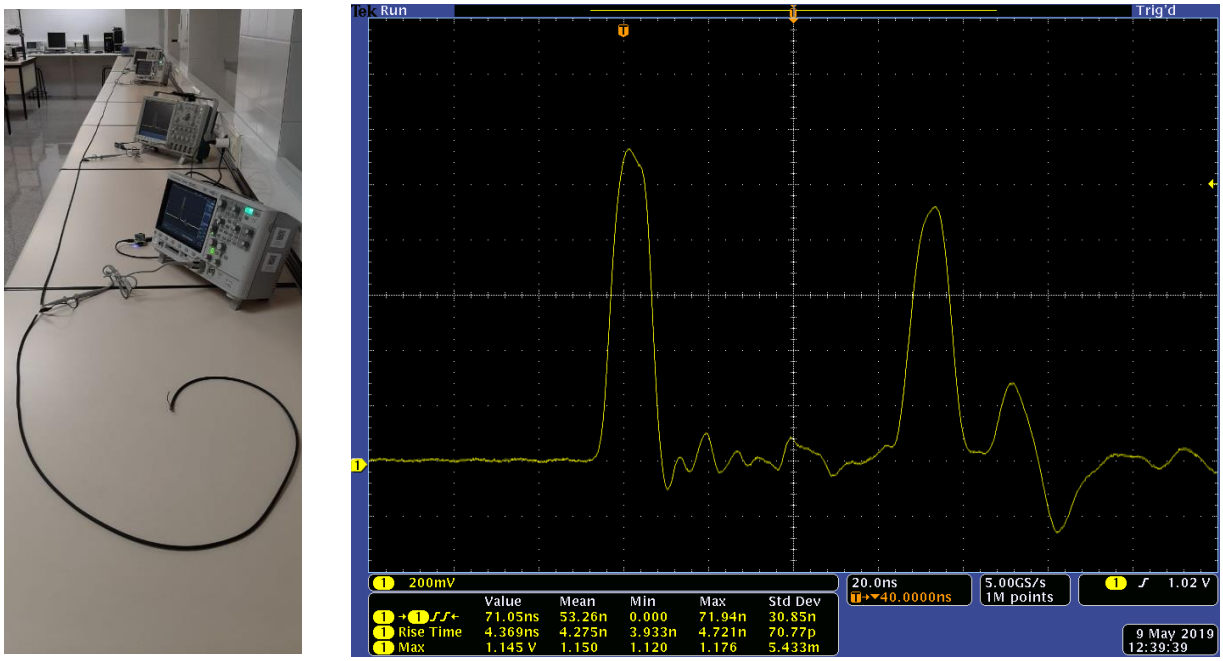

Figura 10: (izda.) Sistema de caracterización del cable coaxial; (dcha.) Captura de pantalla de un osciloscopio conectado a la línea de transmisión coaxial.

De esta forma, con el retraso medido en un punto de la línea y conocida la distancia recorrida calculamos la velocidad de fase de la línea bajo test, dando un valor de $0{,}705c$ (siendo c la velocidad de la luz). La medida obtenida se encuentra dentro de los valores típicos para cables coaxiales [8]. Una vez obtenida la velocidad de propagación del cable, sería posible determinar la distancia desde cualquier punto del cable hasta cualquier otro a partir del retraso entre las señales inicial y reflejada.

Otra posibilidad estudiada consiste en caracterizar el decaimiento de la amplitud de una señal al propagarse por el cable coaxial en función de la distancia. Sabemos de teoría de la señal [8] que la amplitud debería decaer exponencialmente con la distancia, por lo que a partir de un ajuste seremos capaces de determinar la distancia recorrida por una señal entre dos puntos cualesquiera. En este caso, hemos caracterizado el cable midiendo

[3] La posibilidad más sencilla consiste en disponer de una muestra del cable coaxial sobre la que realizar la medida de velocidad de propagación, y aplicar el resultado en el sistema bajo test.

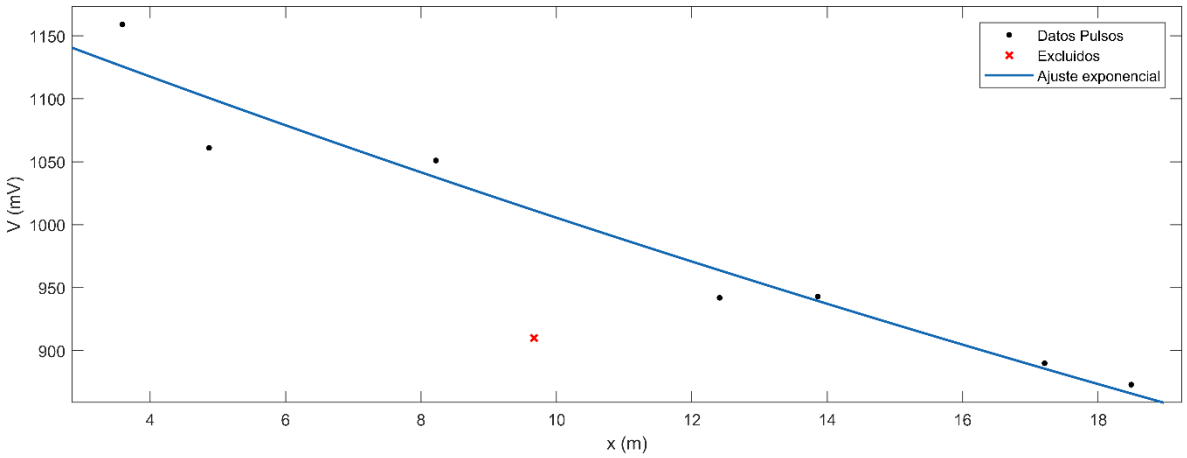

Figura 11: Atenuación del cable coaxial

experimentalmente las amplitudes de la señal procedente del generador y las correspondientes señales reflejadas en varios puntos utilizando los dos DPO4104. De nuevo las medidas con los osciloscopios de las amplitudes de señal y la configuración del generador se han realizado conectando a los instrumentos los módulos inalámbricos descritos en la Sección anterior. Las medidas obtenidas de amplitud frente a la distancia recorrida desde el origen de la señal y su ajuste se presentan en la Figura 11. Cabe comentar que uno de los puntos, el señalado en rojo, ha sido descartado. Creemos que al estar cerca del extremo final de la línea (11,04 m) se ve afectado por la interacción que se forma en este punto entre señal incidente y reflejada, dando un valor más bajo del que debería si la línea fuera más larga. A partir del ajuste de los puntos experimentales obtenemos que la constante de atenuación tiene un valor de 0,018 m^{-1}, que se encuentra dentro de los valores que se manejan para cables coaxiales de este tipo.

4.3 Aplicación del módulo inalámbrico de control con alimentación por batería

Un sistema de control de medida con protocolo inalámbrico permite implementar este tipo de solución en entornos donde los sistemas por cable no son viables o difíciles de mantener, e incluso en ausencia de red eléctrica, empleando instrumentos alimentados con batería. Para analizar su uso en un entorno de estas características, se ha estudiado su operación alimentando uno de los servidores Raspberry con una batería de reducidas dimensiones y valores nominales de 3,7 V y 2000 mAh. Este módulo se ha configurado para controlar la operación de un multímetro digital Agilent 34461A que mide la evolución de la tensión de salida de la propia batería que alimenta el módulo local Raspberry de control. Con esto, seremos capaces de determinar la energía que es capaz de almacenar la batería e intentar estimar cuál es el tiempo de funcionamiento de la terminal.

Dado que cada instrumento tiene un tiempo de respuesta diferente, el consumo por cada instancia (instrucción o pregunta al instrumento) puede variar. Por ello, previamente debemos modelar la corriente que consume la Raspberry Pi Zero en una petición `query` y/o `write` para, en nuestro caso, el multímetro. Dado que el objetivo de esta parte es dar un ejemplo de uso a nuestro sistema y caracterizar la batería, nos vamos a limitar a estudiar el consumo dinámico de corriente de la batería ante el envío de solicitudes de medida de

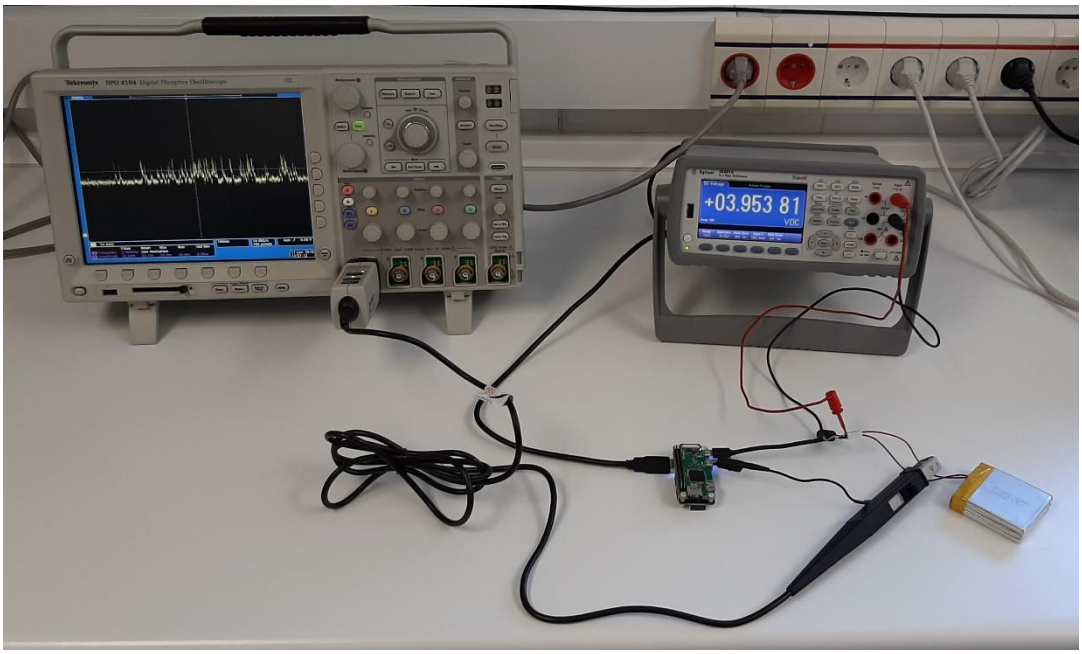

Figura 12: Sistema de caracterización del consumo de la batería

tensión instantánea (pregunta) por parte del módulo de control local al propio multímetro en un proceso automatizado con control no local. El perfil de corriente suministrada al módulo local ha sido obtenido mediante una sonda de efecto Hall Tektronix TCP0030 conectada al osciloscopio Tektronix DPO4104.

El perfil de consumo de corriente de un comando `query` enviado al instrumento se presenta en la Figura 13. Antes y después del comando la batería proporciona al módulo de control un nivel de corriente de base promedio, I_b (A, Figura 13). Cuando el cliente solicita una medida, el perfil de corriente suministrada consta de tres partes asociadas a la recepción del requerimiento desde el PC cliente (B), envío de la instrucción al instrumento y recepción de la respuesta (C) y transmisión de ésta al PC cliente (D), con un promedio I_q.

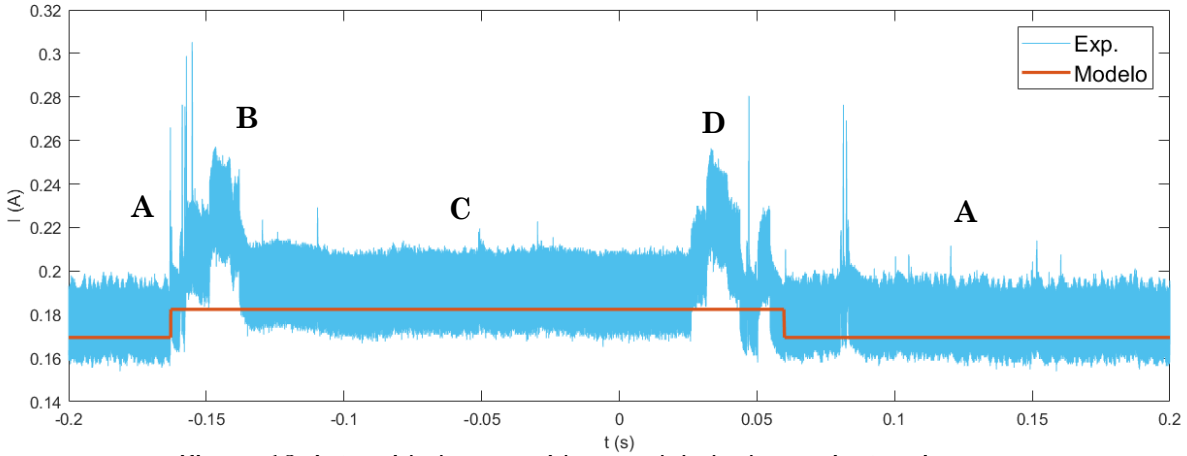

Figura 13: Intensidad consumida y modelada de una instancia `query`

Una vez obtenidos los valores de intensidad, veamos cómo vamos a determinar la energía que puede almacenar la batería. Para ello la conectamos totalmente cargada a la Raspberry Pi Zero y monitorizamos su tensión de salida con el multímetro. Desde el PC de usuario lanzamos el programa `consumo.py` (Anexo A6) que hace que la Raspberry mida el voltaje de la batería y se lo devuelva al usuario cada 10 segundos en bucle infinito, de forma que cuando la batería se agote, la Raspberry se apagará. Obtenemos la curva de voltaje de la Figura 14.

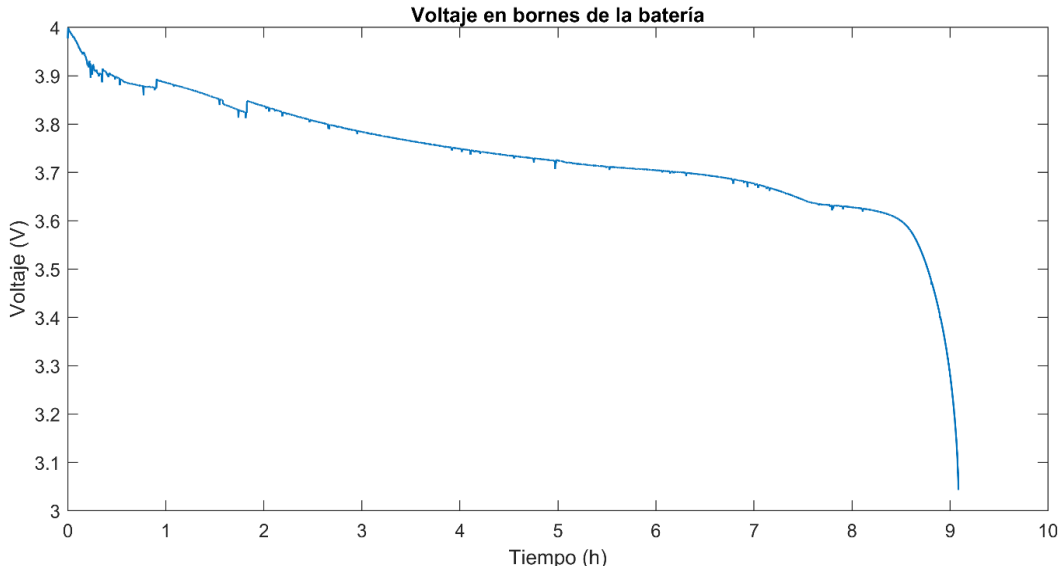

Figura 14: Variación del voltaje de la batería en función del tiempo

Así pues, con esta curva de tensión y los niveles de intensidad sabemos cómo varía la potencia consumida en función del tiempo y, con esto, podemos estimar la energía almacenada por la batería. Para ello vamos a considerar que durante los 10 s que dura el intervalo entre medida y medida, el voltaje es constante. Por otro lado, tenemos que durante un pequeño periodo de tiempo en el intervalo de estos 10 segundos (el que dura la ejecución de la instrucción enviada por el cliente), la intensidad suministrada está al nivel que hemos caracterizado previamente (I_q) y que, durante el resto del tiempo la intensidad es el nivel bajo (I_b). De esta forma obtenemos la curva de potencia, que podemos integrar por trapecios, obteniendo que la energía total que hemos consumido de la batería es de 6,735 W h.

Si comparamos este resultado con los datos nominales de la batería, que son 2000 mAh a 3,7 V, tenemos que la energía nominal que puede almacenar es de 7,4 W h, por lo que creemos que nuestro resultado es bastante razonable teniendo en cuenta que los datos ofrecidos por la batería tienen cierta deriva estadística y que además la batería ya había sido usada con anterioridad y por lo tanto es muy posible que tenga pérdidas de capacidad de almacenaje energético.

Así mismo, a partir de los datos de intensidad por instancia (I_b e I_q) y el voltaje nominal de la batería ($\bar{V}$) podemos estimar dinámicamente el tiempo de vida útil restante de nuestro sistema, conociendo la energía que es capaz de almacenar la batería. Sabemos que una petición query va a consumir la siguiente cantidad de energía (E_i):

$$E_i = (I_b(\nu^{-1} - t_q) + I_q t_q)\bar{V} \tag{2}$$

Donde t_q es el tiempo que dura la instancia `query` y ν la frecuencia con la que se repite, que en el caso de la medida de los voltajes ha sido 0,1 Hz (un `query` cada 10 s). En un caso más general en el que tengamos n_q `query`'s y n_w `write`'s que se repiten con una frecuencia ν, la expresión para la energía consumida por periodo sería:

$$E_i = \left(I_b\left(\nu^{-1} - n_q t_q - n_w t_w\right) + (n_q t_q + n_w t_w)I_q\right)\bar{V} \tag{3}$$

Con todo esto podemos estimar el tiempo máximo como:

$$t_{max} = \frac{E}{E_i \cdot \nu} \tag{4}$$

Donde E es la máxima energía que puede almacenar la batería. Si hacemos este cálculo para nuestra batería usando el valor de E que hemos obtenido antes, para el ejemplo que hemos llevado a cabo la duración estimada de la batería es de 9,17 h, un valor bastante cercano a las 9,08 h reales que el sistema estuvo midiendo antes de que se apagase la Raspberry.

5. Conclusiones y líneas futuras

En el presente trabajo hemos diseñado e implementado un sistema para el control de instrumentación a través de comunicación inalámbrica, puesto a prueba en tres escenarios cuyo uso tenía diferente justificación.

El primer caso (caracterización de los ocho filtros pasabanda) verifica que es posible controlar la instrumentación necesaria para caracterizar varios sistemas aislados, algo que sería de gran interés para departamentos de calidad de industria a la hora de comprobar los productos, por ejemplo. En la caracterización del cable coaxial hemos demostrado que nuestro sistema soluciona el problema que lo inspiró: la capacidad de controlar instrumentación a distancia para poder caracterizar sistemas de gran tamaño. Finalmente, hemos visto como nuestro sistema podría estar alimentado por una batería y cómo podríamos estimar el tiempo de vida de esta en función a las tareas que va a recibir periódicamente. Esta parte abre la puerta a un uso portátil del sistema, dotándolo de una característica extra que podría ser de interés en entornos de medida fuera del laboratorio.

Los objetivos propuestos eran que fuera un sistema integral, escalable y de bajo coste; veamos si nuestro trabajo satisface dichos objetivos:

- Es un sistema integral y genérico: una terminal permite controlar cualquier instrumento que tenga conexión USB de forma inalámbrica como si la estuviésemos controlando a través de un cable USB, sin necesidad de nada más que la Raspberry Pi Zero, el hub USB y el cable de alimentación.
- Es escalable: permite controlar tantas terminales como queramos siempre que haya posibilidad de asignarles IP y puerto para la comunicación WiFi. Así mismo, una misma terminal es capaz de controlar tantos instrumentos como se quiera hasta que el puerto USB se sature por exceso de conexiones.
- Es de bajo coste: el precio de una Rasperry Pi Zero W es de 10,53 € [9] y el del hub Zero4U es de 9,95 € [10], por lo que el precio total del sistema es poco más de 20 €.

Por último, nos gustaría comentar algunas ideas para poder ampliar este trabajo. Por un lado, a nivel de software, hemos desarrollado una librería en Python, `instin.py`, que incluye las funciones necesarias para poder desarrollar programas en este mismo lenguaje para el control de instrumentación. Creemos que podría ser interesante el desarrollo de una librería en Matlab análoga a `insitn.py` que permitiera el uso desde Matlab.

Por otro lado, a nivel de hardware, la Raspberry Pi Zero cuenta con unas especificaciones que sobrepasan las necesarias para las tareas a las que hemos dedicado la placa. Por ello pensamos que una opción para poder reducir el precio y el tamaño del sistema sería buscar un microcontrolador con conectividad WiFi y compatibilidad con el puerto USB en el que incluir el software necesario para que realizase las mismas funciones que nuestro trabajo. Una opción que creemos bastante plausible sería el uso de la placa ESP32 de Espressif [11], la cual cuenta ya con conectividad WiFi de serie y es muy usada hoy en día para aplicaciones IoT (*Internet of the Things*). Eso sí, dado que se trata de un microcontrolador y no de un ordenador como tal, sería necesario desarrollar los *backends* necesarios para la comunicación con la instrumentación a través del protocolo VISA, algo que en nuestro caso lo está realizando el paquete `pyvisa-py` de Python.

Bibliografía

[1] IEEE, 488.2-1987 - IEEE Standard Codes, Formats, Protocols, and Common Commands. for Use With ANSI/IEEE Std 488.1-1987 IEEE Standard Digital Interface for Programmable Instrumentation, 1987.

[2] W. R. Stevens, TCP/IP illustrated. Vol. 1, The protocols, 20 ed., Addison-Wesley, 2001.

[3] W. R. Stevens y G. R. Wright, TCP/IP illustrated. Vol. 2, The implementation, 2 ed., Addison-Wesley, 1995.

[4] Microsoft Corporation, «Microsoft Technet: Virtual Private Networking: An Overview,» [En línea]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/bb742566(v=technet.10). [Último acceso: 2019 mayo 24].

[5] LogMeIn, Inc., «LogMeIn Hamachi Getting Started Guide,» [En línea]. Available: https://documentation.logmein.com/documentation/EN/pdf/Hamachi/LogMeIn_Hamachi_GettingStarted.pdf. [Último acceso: 24 mayo 2019].

[6] UUGear s.r.o., «Zero4U User Manual,» [En línea]. Available: http://www.uugear.com/doc/Zero4U_UserManual.pdf. [Último acceso: 24 mayo 2019].

[7] Everett Charles Technologies, «Spring Probes,» [En línea]. Available: https://ect-cpg.com/spring-probes. [Último acceso: 24 mayo 2019].

[8] W. C. Johnson, Trasnmission Lines and Networks, International Student ed., McGraw-Hill Kogakusha, Ltd., 1950.

[9] KUBII, «KUBII (distribuidor oficial de Raspberry en España),» [En línea]. Available: https://www.kubii.es/pi-zero-w/1851-raspberry-pi-zero-w-kubii-3272496006997.html. [Último acceso: 17 mayo 2019].

[10] Adafruit Industries, LLC, «Adafruit Shop,» [En línea]. Available: https://www.adafruit.com/product/3298. [Último acceso: 17 mayo 2019].

[11] Espressif Systems, «ESP32 by Espressif,» [En línea]. Available: https://www.espressif.com/en/products/hardware/esp32/overview. [Último acceso: 17 mayo 2019].

[12] Keysight Technologies, Inc., Keysight InfiniiVision 2000 X-Series Oscilloscopes Programmer Manual.

[13] Tektronix, Inc., MSO4000 and DPO4000 Series Digital Phosphor Oscilloscopes Programmer Manual.

[14] Tektronix, Inc., AFG3000 Series Arbitrary/Function Generators Programmer Manual.

[15] Keysight Technologies, Inc., Keysight Truevolt Series Digital Multimeters Operating and Service Guide.

[16] Tektronix, Inc., MSO/DPO5000, DPO7000/C, DPO70000/B/C/D, DSA70000/B/C/D, and MSO70000/C Series Digital Oscilloscopes Programmer Manual.

Anexos

A1. Programa *bash* para la instalación de paquetes de Python

```
#!/bin/bash

echo "Comenzando la instalación de los paquetes de Python 3.5......\n"
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install pip
echo "Asistentes instalación actualizados.....................\n"

echo "Instlando paquetes VISA (pyvisa y pyvisa-py).............\n"
sudo pip3 install python3-pyvisa
sudo pip3 install python3-pyvisa-py
echo "Paquetes VISA instalados (pyvisa y pyvisa-py)............\n"

echo "Instalando paquetes matemáticos (numpy y matplotlib)...\n"
sudo pip3 install python3-numpy
sudo pip3 install python3-matplotlib
echo "Paquetes matemáticos instalados (numpy y matplotlib)..\n"

echo "Instlando paquete control USB...........................\n"
sudo pip3 install pyusb
echo "Paquete control USB instalado...........................\n"

echo "Reiniciando.....................................\n"

sudo reboot
```

A2. Librería `instin.py`

```
import socket
import sys
import os
import time

t_ret = 0.05 #Tiempo entre conexión TCP y conexión TCP
BUFF = 1024*8

def relleno_p2(path):
    file = open(path, 'rb')
    file_len = len(file.read())
    file.close()
```

```
    new_file_len = 2

    while new_file_len < file_len:
        new_file_len = new_file_len*2

    while file_len < new_file_len:
        file = open(path, 'ab')
        file.write(b'\n')
        file.close()

        file = open(path, 'rb')
        file_len = len(file.read())
        file.close()

    return new_file_len




def sincro(HOST, PORT, msg):
    """
    Sirve para poner en hora las terminales a las que te quieres conectar

    """
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    #print('Socket created')

    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    s.close() #Cerramos la comunicación

    time.sleep(t_ret)
```

```
def write(HOST, PORT, n_inst, msg):
    """
    Le debemos pasar la ip del terminal,
    el puerto, el número de instrumento
    asociado al instrumento que queremos
    usar en esa terminal y el mensaje.

    Si queremos cerrar el programa (y reiniciar las raspi) debemos mandar,
    con cualquier n_inst diferente de 0, 'close' en msg.
    """
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    #print('Socket created')

    msg = str(n_inst) + msg
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    s.close() #Cerramos la comunicación

    time.sleep(t_ret)

def query(HOST, PORT, n_inst, msg):
    """
    Le debemos pasar la ip del terminal,
    el puerto, el número de instrumento
    asociado al instrumento que queremos
    usar en esa terminal y el mensaje.
    """
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    #print('Socket created')

    msg = str(n_inst) + msg
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío del mensaje

    #s.close() #Cierre de la conexión

    time.sleep(t_ret)

    data = s.recv(BUFF)
    mensaje = data.decode('ASCII')

    s.close() #Cierre de la conexión

    time.sleep(t_ret)

    return mensaje

def open_inst(HOST, PORT, msg):
    """
    Si queremos abrir el instrumento tenemos que en msg
    escribir la dirección por ejemplo:
    'USB0::0x0957::0x179B::MY51250760::INSTR'
    """
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
```

```
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    #print('Socket created')

    msg = '0' + msg
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    time.sleep(t_ret)

    print('Instrumento ' + msg + ' abierto en ' + HOST)

    data = s.recv(BUFF)
    mensaje = data.decode('ASCII')

    s.close() #Cierre de la conexión

    time.sleep(t_ret)

    return mensaje

def close_term(HOST, PORT):
    """
    Si queremos cerrar el programa con cualquier n_inst
    diferente de 0 mandamos 'close' en msg.
    """
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    #print('Socket created')

    msg = '1' + 'close'
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    s.close() #Cerramos la cominación
```

```
    print('Terminal ' + HOST + ' cerrada.')
    time.sleep(t_ret)

def send_program(HOST, PORT, NAME_IN, *NAMEs_OUT):
    #Creamos la conexión
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except socket.error as err:
        print('Socket creation error:')
        print(err)
        sys.exit()

    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo
puerto
    s.connect((HOST, PORT)) #Nos conectamos al servidor
    print('Socket created')

    #Enviamos el path de entrada
    msg = 'p/home/pi/Desktop/' + NAME_IN
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    time.sleep(t_ret)

    #Devuelve el path de entrada
    data = s.recv(BUFF)
    print(data.decode('ASCII'))

    #Enviamos el numero de paths de salida
    msg = str(len(NAMEs_OUT))
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío

    #Devuelve el numero paths de salida
    data = s.recv(BUFF)
    print(data.decode('ASCII'))

    fout=[]
    #Enviamos los paths del programa de salida y los abrimos aquí
    for name in NAMEs_OUT:
        fout.append(open(name, 'wb'))
        msg = '/home/pi/Desktop/' + name
```

```
    data = msg.encode('ASCII') #Pasamos a tipo byte
    s.send(data) #Envío
    time.sleep(t_ret)

    #Devuelve el path de salida
    data = s.recv(BUFF)
    print(data.decode('ASCII'))
    time.sleep(t_ret)

time.sleep(1)

#Enviamos el tamaño del programa de entrada
lenght = relleno_p2(NAME_IN)
print(lenght)
msg = str(lenght)
data = msg.encode('ASCII') #Pasamos a tipo byte
s.send(data)

#Devuelve lo mismo
data = s.recv(BUFF)
print(data.decode('ASCII'))

time.sleep(1)

#Enviamos el programa de entrada
f = open(NAME_IN, 'rb')
print(f.read(lenght))
f.close()
f = open (NAME_IN, "rb")

s.send(f.read(lenght))
f.close()

data = s.recv(BUFF)
print(data.decode('ASCII'))

time.sleep(1)



#Recibimos los programas de salida
for k in range(len(NAMEs_OUT)):
```

```
        msg = 'PC: Listo para recibir tamaño archivo'
        data = msg.encode('ASCII') #Pasamos a tipo byte
        s.send(data)

        #Recibimos primero el tamaño
        data = s.recv(512)
        msg = data.decode('ASCII')
        print(msg)
        n_paq = int(float(msg))
        print('Numero de paquetes a recibir ' + str(n_paq))

        msg = 'PC: Listo para recibir archivo'
        data = msg.encode('ASCII') #Pasamos a tipo byte
        s.send(data)

        for h in range(n_paq):
            data = s.recv(512)
            #print(data)
            fout[k].write(data)
        fout[k].close()
        time.sleep(1)

    s.close()
```

A3. Programa autorun.py

```
import os
import visa # librería de control de instrumentación
import socket
from time import sleep

port = 50001

N_enlaces = 5
BUFF = 1024*8

def relleno_p2(path):
    file = open(path, 'rb')
    file_len = len(file.read())
    file.close()

    new_file_len = 512

    while new_file_len < file_len:
```

```
        new_file_len = new_file_len + 512

    file = open(path, 'ab')
    for k in range(new_file_len - file_len):
        file.write(b'\n')
    file.close()

    return new_file_len

def mandar_mensaje(conexion, msg):
    data = msg.encode('ASCII') #A tipo byte
    conexion.send(data) #Envío

def recibir_mensaje(conexion):
    #Recibimos el mensaje
    data = conexion.recv(BUFF)
    mensaje = data.decode('ASCII')

    return mensaje

def launch_program(conexion, path_in):
    #Primero recibimos el programa a ejecutar
    fin = open(path_in,'wb') #Lo abrimos en binario

    n_paths_out = recibir_mensaje(conexion)
    print('Número de archivos de salida: '+n_paths_out)
    mandar_mensaje(conexion, 'Archivos de salida '+ n_paths_out)

    paths_out = []

    for pout in range(int(n_paths_out)):
        paths_out.append(recibir_mensaje(conexion))
        mandar_mensaje(conexion, 'Archivo de salida '+str(pout)+':'+paths_out[pout])

    #Recibimos el tamaño del programa de salida
    fin_lengh = recibir_mensaje(conexion)
    print(int(fin_lengh))
    print('Tamaño programa entrada ' + fin_lengh)

    mandar_mensaje(conexion, 'El tamaño del programa de entrada es '+fin_lengh)

    print('Recibimos el programa a ejecutar')
```

```
n_paq = int(fin_lengh)/512
print(n_paq)
for k in range(int(n_paq)):
    data = conexion.recv(512)
    print(data)
    fin.write(data)
fin.close()

mandar_mensaje(conexion, 'Programa recibido')

print('Programa iniciándose')
os.system('sudo python3 '+path_in) #Lanzamos el programa
print('Fin de ejecución del programa, devolviendo archivos de salida')

for pout in range(int(n_paths_out)):
    print(recibir_mensaje(conexion))

    leng = relleno_p2(paths_out[pout])
    print('Tamaño programa salida ' + str(pout) + ': '+ str(leng))

    #Enviamos el numero de paquetes de 512 b de los programas de salida
    n_paq = int(leng)/512.0
    mandar_mensaje(conexion, str(n_paq))
    print(n_paq)

    file_out = open(paths_out[pout],'rb')

    print(recibir_mensaje(conexion))

    for k in range(int(n_paq)):
        data = file_out.read(512)
        print(data)
        conexion.send(data)
        sleep(0.2)

    file_out.close()

    sleep(1)


print('Datos enviados')
```

```
rm=visa.ResourceManager('@py')

#Creamos el objeto socket:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) #Nos permite usar el mismo puerto
try:
    s.bind(('', port))
except socket.error as err:
    print('Bind failed')
    print(err)
s.listen(N_enlaces) #Limitamos el número de clientes que se pueden conectar al servidor
print('Socket awaiting messages')

(conn, addr) = s.accept() #Nos conectamos
print('Connected to ' + str(addr))
msg = recibir_mensaje(conn)
os.system('sudo date ' + msg) #Ponemos en hora la Raspi
conn.close()

print('Inicio programa')

inst = []
inst.append(0)

while True:
    (conn, addr) = s.accept() #Nos conectamos
    print('Connected to ' + str(addr))

    msg = recibir_mensaje(conn) #Leemos el mensaje
    print(msg)

    if msg[0] == 'p':
        mandar_mensaje(conn, 'Abriendo '+msg[1:])
        launch_program(conn, msg[1:])
    else:
        n_inst = int(msg[0])
        msg = msg[1:]

        if n_inst == 0: #Si pasamos un cero es porque abrimos nuevo instrumento
            inst.append(rm.open_resource(msg))
            mandar_mensaje(conn, str(len(inst)-1)) #devolvemos el ID que le asigna la RasPi
al nuevo instrumento
```

```
        elif '?' in msg: #Si el mensaje tiene ? es query
            data = inst[n_inst].query(msg)
            mandar_mensaje(conn, str(data))

        elif msg == 'close': #Si el mensaje es close apagamos
            conn.close()
            s.close()
            os.system('reboot')

        else: #Ya sólo nos queda que sea un write
            inst[n_inst].write(msg)

    conn.close()
```

A4. Programa para la caracterización de dos filtros pasabanda `puesto_1.py`

Cabe comentar que los demás `puesto_i.py` son análogos sólo que, cambiando el título de los archivos de salida para evitar sobre escrituras y las direcciones VISA de los instrumentos a controlar. Los comandos SCPI los hemos buscado en [12].

```
import visa
import time
import numpy as np
import matplotlib.pyplot as plt


titulo="/home/pi/Desktop/data_p1_osc" #data_pi_osc, i indica el número de
puesto
VISA = ['USB0::0x0957::0x179B::MY51250756::INSTR',
'USB0::0x0957::0x179B::MY51250652::INSTR'] #Cambiar en cada puesto


pasos=100 #Número de puntos en frecuencia a tomar


freq=np.logspace(np.log10(15000),np.log10(25000),pasos) #Elegir las frecuencias de inicio y
final para el barrido


resources=visa.ResourceManager('@py') #Abrimos el gestor de instrumentos


N_instr = len(VISA)


medida_i=np.zeros(N_instr*pasos) #Inicializamos las variables donde vamos a guardar los
datos
medida_o=np.zeros(N_instr*pasos)
fase=np.zeros(N_instr*pasos)
```

```
instrumento=[]
for j in range(N_instr): instrumento.append(resources.open_resource(VISA[j])) #Abrimos los
intrumentos


print('Configurando salida del generador de ondas y encendiendo canales')
for j in range(N_instr): instrumento[j].write('wgen:freq '+str(freq[0]))
for j in range(N_instr): instrumento[j].write('wgen:func sin;volt 1;volt:offs 0')
for j in range(N_instr): instrumento[j].write('wgen:outp 1')
for j in range(N_instr): instrumento[j].write('chan1:disp 1;coup dc')
for j in range(N_instr): instrumento[j].write('chan2:disp 1;coup dc')
for j in range(N_instr): instrumento[j].write('autoscale')
for j in range(N_instr): instrumento[j].write('chan1:disp 1;coup dc')
for j in range(N_instr): instrumento[j].write('chan2:disp 1;coup dc')
plt.pause(1)


print('Ajustando canales')
for j in range(N_instr): instrumento[j].write('chan1:offs 0;:chan2:offs 0')
for j in range(N_instr): instrumento[j].write('trig:edge:sour chan1')
for j in range(N_instr): instrumento[j].write('acq:count 8;type aver')
for j in range(N_instr): instrumento[j].write('tim:rang '+str(5/freq[0]))
for j in range(N_instr): instrumento[j].write('chan1:rang 2')
time.sleep(1)
for j in range(N_instr):
       print('Canal ' + str(j))
       vamp=float(instrumento[j].query('meas:vamp? chan2'))
       time.sleep(0.1)
       instrumento[j].write('chan2:rang '+str(5*vamp))


print('Comenzando a medir')
for i in range(pasos):
    print('Medida ' + str(i) + '/' + str(pasos))
    for j in range(N_instr): instrumento[j].write('tim:rang '+str(5/freq[i]))
    for j in range(N_instr): instrumento[j].write('wgen:freq '+str(freq[i]))
    for j in range(N_instr):
       medida_i[i + j*pasos]=float(instrumento[j].query('meas:vamp? chan1'))
       time.sleep(0.05)
    for j in range(N_instr):
       medida_o[i + j*pasos]=float(instrumento[j].query('meas:vamp? chan2'))
       time.sleep(0.05)
    for j in range(N_instr): fase[i + j*pasos]=float(instrumento[j].query('meas:phas?
chan2,chan1'))
    for j in range(N_instr): instrumento[j].write('chan2:rang '+str(medida_o[i +
j*pasos]*1.5))
```

```
    plt.pause(0.2)


print('Cerrando los instrumentos')
for j in range(N_instr): instrumento[j].close() #Cerramos los instrumentos


print('Escribiendo archivos de salida')
for j in range(N_instr):
       fout = open(titulo+str(j)+".txt", 'w+')
       fout.write("VISA:"+VISA[j]+"\nHora:"+ time.strftime('%m%d%H%M.%S')+"\nFreq(Hz)
Vin(V) Vout(V) Phase(deg)\n")
       for i in range(pasos): fout.write("%5.2f\t%5.2f\t%5.2f\t%5.2f\n" % (freq[i],
medida_i[i+j*pasos], medida_o[i+j*pasos], fase[i+j*pasos]))
       fout.close()
```

A5. Programa para la medida del retraso de una señal y la atenuación en un cable coaxial (`coaxial.py`)

Hemos usado los manuales del DPO 4004 [13] y el AFG 3252 [14] para encontrar los comandos SCPI necesarios.

```
import instin as i
import time


#Parámetros del generador:
freq_ret = '1e6'
duty = '1'
vpp = '1'
offset = '500mV'


#Datos de las terminales para la conexión remota
ip_gen = '192.168.1.4' #Generador de ondas
port_gen = 50001
ip_osc0 = '192.168.1.7' #Osciloscopio a 3.54 m
port_osc0 = 50002
ip_osc1 = '192.168.1.6' #Osciloscopio a más de 3.54 m
port_osc1 = 50003


gen = 'USB0::0x0699::0x0345::C022028::INSTR'
osc0 = 'USB0::0x0699::0x0401::C022342::INSTR'
osc1 = 'USB0::0x0699::0x0401::C022329::INSTR'


#Inicialización de la terminal
print('Hora PC: ' + time.ctime(time.time()))
i.sincro(ip_gen, port_gen, time.strftime('%m%d%H%M.%S'))
i.sincro(ip_osc0, port_osc0, time.strftime('%m%d%H%M.%S'))
```

```
i.sincro(ip_osc1, port_osc1, time.strftime('%m%d%H%M.%S'))

#Abrimos los instrumentos
print('Abriendo instrumentos')
gen = int(i.open_inst(ip_gen, port_gen, gen))
osc0 = int(i.open_inst(ip_osc0, port_osc0, osc0))
osc1 = int(i.open_inst(ip_osc1, port_osc1, osc1))

#Ajustamos generador de ondas a la frecuencia deseada
print('Ajustando el generador de ondas')
i.write(ip_gen, port_gen, gen, 'func puls')
i.write(ip_gen, port_gen, gen, 'freq '+ freq_ret)
i.write(ip_gen, port_gen, gen, 'volt:ampl '+ vpp + ' vpp')
i.write(ip_gen, port_gen, gen, 'volt:offs '+ offset)
i.write(ip_gen, port_gen, gen, 'puls:dcyc '+ duty) #Ponemos el duty cycle al 1%
i.write(ip_gen, port_gen, gen, 'outp 1')
time.sleep(0.5)

#Ajustamos los osciloscopios
print('Ajustamos los osciloscopios')
t_duty = (float(duty)/100)*(1/float(freq_ret))
i.write(ip_osc0, port_osc0, osc0, 'autos exec')
i.write(ip_osc1, port_osc1, osc1, 'autos exec')
time.sleep(2)
i.write(ip_osc0, port_osc0, osc0, 'trig:a:lev 0.95') #Sabemos que la amplitud del pulso
sale ocn V
i.write(ip_osc1, port_osc1, osc1, 'trig:a:lev 0.95')
i.write(ip_osc0, port_osc0, osc0, 'hor:del:tim ' + str(4*t_duty))
i.write(ip_osc1, port_osc1, osc1, 'hor:del:tim ' + str(4*t_duty))
i.write(ip_osc0, port_osc0, osc0, 'hor:sca ' + str(2*t_duty))
i.write(ip_osc1, port_osc1, osc1, 'hor:sca ' + str(2*t_duty))
i.write(ip_osc0, port_osc0, osc0, 'ch1:pos -3') #Sabemos que es una señal digital, la
bajamos hasta abajo
i.write(ip_osc1, port_osc1, osc1, 'ch1:pos -3')
i.write(ip_osc0, port_osc0, osc0, 'ch1:sca 200e-3')
i.write(ip_osc1, port_osc1, osc1, 'ch1:sca 200e-3')
i.write(ip_osc0, port_osc0, osc0, 'acq:mod ave')
i.write(ip_osc1, port_osc1, osc1, 'acq:mod ave')
i.write(ip_osc0, port_osc0, osc0, 'acq:numav 8')
i.write(ip_osc1, port_osc1, osc1, 'acq:numav 8')
time.sleep(0.5)

print('Ajustamos las medidas en los osciloscopios')
```

```
i.write(ip_osc0, port_osc0, osc0, 'measu:meas1:del:dir backw') #para que coja el segundo
pico
i.write(ip_osc1, port_osc1, osc1, 'measu:meas1:del:dir backw')
i.write(ip_osc0, port_osc0, osc0, 'measu:meas1:type del')
i.write(ip_osc1, port_osc1, osc1, 'measu:meas1:type del')
i.write(ip_osc0, port_osc0, osc0, 'measu:meas2:type ris')
i.write(ip_osc1, port_osc1, osc1, 'measu:meas2:type ris')
i.write(ip_osc0, port_osc0, osc0, 'measu:meas3:type max')
i.write(ip_osc1, port_osc1, osc1, 'measu:meas3:type max')
time.sleep(0.5)


print('Midiendo la altura del pico que va')
#Nos va a permitir elegir los niveles de referencia para la medida del tiempo de subida y
del retardo
altp1o0 = float(i.query(ip_osc0, port_osc0, osc0, 'measu:meas3:mean?'))
altp1o1 = float(i.query(ip_osc1, port_osc1, osc1, 'measu:meas3:mean?'))
time.sleep(0.5)


print('Estableciendo niveles de referencia')
i.write(ip_osc0, port_osc0, osc0, 'measu:refl:meth abs')
i.write(ip_osc1, port_osc1, osc1, 'measu:refl:meth abs')
i.write(ip_osc0, port_osc0, osc0, 'measu:refl:abs:high '+str(altp1o0*0.9))
i.write(ip_osc1, port_osc1, osc1, 'measu:refl:abs:high '+str(altp1o1*0.9))
i.write(ip_osc0, port_osc0, osc0, 'measu:refl:abs:low '+str(altp1o0*0.1))
i.write(ip_osc1, port_osc1, osc1, 'measu:refl:abs:low '+str(altp1o1*0.1))
i.write(ip_osc0, port_osc0, osc0, 'measu:refl:abs:mid1 '+str(altp1o0*0.5))
i.write(ip_osc1, port_osc1, osc1, 'measu:refl:abs:mid1 '+str(altp1o1*0.5))
i.write(ip_osc0, port_osc0, osc0, 'measu:refl:abs:mid2 '+str(altp1o0*0.5))
i.write(ip_osc1, port_osc1, osc1, 'measu:refl:abs:mid2 '+str(altp1o1*0.5))
time.sleep(0.5)


print('Midiendo el retardo entre pico que va y el que viene')
ret0 = float(i.query(ip_osc0, port_osc0, osc0, 'measu:meas1:mean?'))
ret1 = float(i.query(ip_osc1, port_osc1, osc1, 'measu:meas1:mean?'))
time.sleep(0.5)


print('Midiendo el tiempo de subida del pico que va')
risp1o0 = float(i.query(ip_osc0, port_osc0, osc0, 'measu:meas2:mean?'))
risp1o1 = float(i.query(ip_osc1, port_osc1, osc1, 'measu:meas2:mean?'))
time.sleep(0.5)


print('Centrando el pico de vuelta')
i.write(ip_osc0, port_osc0, osc0, 'hor:del:tim '+str(ret0)) #Centramos le pico de vuelta
```

```
i.write(ip_osc1, port_osc1, osc1, 'hor:del:tim '+str(ret1))
i.write(ip_osc0, port_osc0, osc0, 'hor:sca '+str(0.5*risp1o0)) #Sacamos fuera de pantalla
el de ida
i.write(ip_osc1, port_osc1, osc1, 'hor:sca '+str(0.5*risp1o1))
time.sleep(1)

print('Midiendo la altura del pico de vuelta')
altp2o0 = float(i.query(ip_osc0, port_osc0, osc0, 'measu:meas3:mean?'))
altp2o1 = float(i.query(ip_osc1, port_osc1, osc1, 'measu:meas3:mean?'))

print('Osciloscopio cero')
print('Altura primer pico (V): '+str(altp1o0))
print('Altura segundo pico (V): '+str(altp2o0))
print('Tiempo de retardo entre picos (s): '+str(ret0))



print('Osciloscopio uno')
print('Altura primer pico (V): '+str(altp1o1))
print('Altura segundo pico (V): '+str(altp2o1))
print('Tiempo de retardo entre picos (s): '+str(ret1))



fout = open('coaxial_data.txt', 'w')
fout.write('Osciloscopio cero')
fout.write('\nAltura primer pico (V): '+str(altp1o0))
fout.write('\nAltura segundo pico (V): '+str(altp2o0))
fout.write('\nTiempo de retardo entre picos (s): '+str(ret0))

fout.write('\n\n\n')
fout.write('Osciloscopio uno')
fout.write('\nAltura primer pico (V): '+str(altp1o1))
fout.write('\nAltura segundo pico (V): '+str(altp2o1))
fout.write('\nTiempo de retardo entre picos (s): '+str(ret1))

fout.close()

print('Cerrando terminales')
i.close_term(ip_gen, port_gen)
i.close_term(ip_osc0, port_osc0)
i.close_term(ip_osc1, port_osc1)
```

A6. Programa para la medida del voltaje de la batería que alimenta a la terminal `consumo.py`

Hemos usado el manual [15] para buscar el comando SCPI de medida del voltaje en DC.

```
#Importamos las librerías que vamos a usar.
import instin as i
import time

#Datos de las terminales para la conexión remota
ip = '192.168.1.7' #Osciloscopio a 3.54 m
port = 50001

VISA = 'USB0::0x0957::0x1A07::MY53201065::0::INSTR'

#Inicialización de la terminal
print('Hora PC: ' + time.ctime(time.time()))
i.sincro(ip, port, time.strftime('%m%d%H%M.%S'))

#Abrimos los instrumentos
print('Abriendo instrumentos')
multi = int(i.open_inst(ip, port, VISA))

print('Abriendo archivo')
fout = open('consumo_data2.txt', 'w')
fout.write('Medida del voltaje en bornes de la batería\n')
fout.write('Instrumento: ' + VISA + '\n')
fout.write('Hora de inicio: ' + time.strftime('%H:%M:%S') + '\n\n')
fout.write('Tiempo (s) \t Voltaje (V)\n')

time0 = time.time()

while True:
       fout.write(str(time.time()-time0) + '\t' + i.query(ip, port, multi, 'meas:volt:dc?')
+ '\n')
       time.sleep(10)
```

Lista de Figuras

Figura 1: Esquema del sistema desarrollado 1

Figura 2: (izda.) Esquema del hardware de la Raspberry Pi Zero W; (dcha.) Vista del módulo Zero4U acoplado a una Raspberry Pi Zero W 6

Figura 3: Ejemplo de una conexión TCP entre cliente (izquierda) y servidor (derecha) usando la librería `socket` de Python. El servidor envía un dato al cliente y este se lo devuelve. 10

Figura 4: Ejemplo de una conexión TCP entre un servidor (izquierda) establecido con la librería `socket` de Python y un cliente usando Matlab (derecha). 11

Figura 5: Esquema de las conexiones hardware para la comunicación inalámbrica que incluye las librerías utilizadas (azul) y los códigos desarrollados (rojo). 12

Figura 6: Formato de los mensajes que recibe `autorun.py` 12

Figura 7: Diagrama de flujo de un módulo local y funciones del cliente implicadas (en rojo) 16

Figura 8: (izda.) Sistema de caracterización de 8 filtros en paralelo; (dcha.) detalle de uno de los puestos. 17

Figura 9: Programa `launcher_1.py` 18

Figura 10: (izda.) Sistema de caracterización del cable coaxial; (dcha.) Captura de pantalla de un osciloscopio conectado a la línea de transmisión coaxial. 20

Figura 11: Atenuación del cable coaxial 21

Figura 12: Sistema de caracterización del consumo de la batería 22

Figura 13: Intensidad consumida y modelada de una instancia `query` 22

Figura 14: Variación del voltaje de la batería en función del tiempo 23

Lista de acrónimos

- **ACK**: Acuse de recibo (acknowledgement)
- **ASCII**: American Standard Code for Information Interchange
- **B**: Byte
- **FTP**: File Transfer Protocol
- **GPIB**: General-Purpose Instrumentation Bus
- **HDMI**: High-Definition Multimedia Interface
- **HTTP**: Hypertext Transfer Protocol
- **IANA**: Internet Assigned Numbers Authority
- **IEEE**: Instituto de Ingeniería Eléctrica y Electrónica
- **IoT**: Internet of Things
- **IP**: Internet Protocol
- **MSS**: Maximun Segment Size
- **NI**: National Instruments
- **PC**: Personal Computer
- **PCB**: Printed Circuit Board
- **PCI**: Peripheral Component Interconnect
- **PXI**: PCI eXtensions for Instrumentation
- **SBC**: Single Board Computer
- **SCPI**: Standard Commands for Programmable Instruments
- **SD**: Secure Digital
- **SMTP**: Simple Mail Transfer Protocol
- **SSH**: Secure SHell
- **TCP**: Transmission Control Protocol
- **TFG**: Trabajo Fin de Grado
- **USB**: Universal Serial Bus
- **VISA**: Virtual Instrument Software Architecture
- **VPN**: Virtual Private Network
- **VME**: Versa Module Europa
- **VXI**: VME eXtensions for Instrumentation
- **WiFi**: Wireless Fidelity