



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

MEDIKUNTZA
ETA ERIZAINZA
FAKULTATEA
FACULTAD
DE MEDICINA
Y ENFERMERÍA

1

MASTER EN INGENIERÍA BIOMÉDICA INGENIARITZA BIOMEDIKOA MASTERRA

TRABAJO FIN DE MASTER

**DISEÑO Y DESARROLLO DE UN SISTEMA DE APOYO
A LA DIAGNOSIS E IDENTIFICACIÓN DE
PATOLOGÍAS IMPLEMENTADO CON DISPOSITIVOS
DE BAJO COSTE, PRECISO, FIABLE Y DE USO EN
ENTORNOS DOMICILIARIOS, RESIDENCIALES Y
HOSPITALARIOS.**

Nombre del Alumno/Alumna: Oinatz, Aspiazu, Ituarte

Director/Directora: Eloy, Irigoyen, Gordo

Departamento/Facultad: Departamento de Ingeniería de Sistemas y Automática (GICI - Escuela de Ingenieros de Bilbao) / Facultad de Medicina y Enfermería (UPV-EHU)

Curso académico 2021-2022

Localidad: Bilbo / Bilbao

Fecha de entrega: 19, Septiembre, 2022



Agradecimientos

A Eloy, por darme la oportunidad de trabajar en un proyecto con el que he disfrutado mucho.

A Sandra, mi mujer, por toda la paciencia durante todas las horas de dedicación de este Máster.

A mi hijo Iker, que a pesar de tener poco más de un año, nos proporciona un cariño infinito.

A toda mi familia por estar siempre ahí.



Abstract

A current lack in the health field is the active and remote monitoring of the vital signs of our elders and especially of people who have diseases or are in poor health. Unfortunately, the situation that we have experienced during the last two years due to the pandemic caused by the SARS-CoV-2 virus [1], has made us understand that our health professionals do not have a vision of the state of these groups of people. That is why the proposed active monitoring could also serve not only in emergency situations, but also provides a mechanism to be proactive when detecting arrhythmias or other pathologies in apparently healthy people. It also helps professionals to detect strange patterns in the data obtained and to learn from the changes in vital signs among people catalogued with some type of pathology.

The active monitoring of vital signs carried out remotely also implies that health professionals could have a complete history of the health status of patients for days, months and even years. This extensive information is really valuable since it provides an amount of data which is not known today. In addition, when it is done remotely, the cost for hospitals and human efforts is reduced to the maintenance of computer servers, computer personnel and electronic devices used by the patients.



Sumario

1. INTRODUCCIÓN.....	8
1.1. MOTIVACIÓN.....	8
1.2. OBJETIVOS.....	8
1.3. ESTRUCTURA DEL DOCUMENTO.....	9
2. ESTADO DEL ARTE.....	9
2.1. SENsoRES.....	9
2.1.1. PULSioxIMETRÍA.....	10
2.1.1.1. MAX30100 y MAX30102.....	10
2.1.1.2. GSR (GALVANIC SKIN RESPONSE – RESPUESTA GALVÁNICA DE LA PIEL).....	11
2.2. PUERTO Y PROTOCOLO I2C.....	12
2.3. ANALOG-TO-DIGITAL CONVERTER (ADC) - CONVERSOR ANALÓGICO DIGITAL.....	14
2.3.1 ADS1115.....	15
2.4. DISPOSITIVOS DE COMPUTACIÓN PORTABLES Y DE BAJO COSTE.....	15
2.4.1. ESP32.....	15
2.4.2. RASPBERRY PI ZERO 2 W.....	17
2.4.3. RADXA ZERO.....	18
2.4.4. RASPBERRY PI 400.....	20
2.5. SOFTWARE.....	21
2.5.1. LENGUAJES DE PROGRAMACIÓN.....	21
2.5.1.1. PYTHON.....	21
2.5.1.2. MICROPYTHON.....	21
2.5.1.3 HTMLY JSON.....	22
2.5.2. POSTGRESQL.....	22
2.5.3. ENTORNO DE DESARROLLO.....	22
3. DISEÑO DE LA SOLUCIÓN PROPUESTA.....	23
3.1. ARQUITECTURA DE LA SOLUCIÓN PROPUESTA.....	23
3.2 DIAGRAMA DE SECUENCIA.....	24
4. IMPLANTACIÓN DE LA SOLUCIÓN.....	25



4.1 CONEXIONES E I2C.....	25
4.2 INSTALACIÓN Y MANEJO DE BASE DE DATOS.....	26
4.2.1. CONFIGURACIÓN RASPBERRY PI.....	26
4.2.2. CONFIGURACIÓN DE SERVIDOR DE BASE DE DATOS.....	26
4.2.3. INSERTS EN BASE DE DATOS.....	28
4.2.3.1. READ_SEQUENTIAL().....	28
4.2.3.2. PSYCOPG2.EXTRAS.EXECUTE_BATCH.....	29
4.2.3.3. CIFRADO CONTRASEÑA BASE DE DATOS.....	29
4.3. ADC ADS1115.....	30
4.4. OBTENER_ID().....	30
4.5. MULTIPROCESO.....	31
4.5.1. POOL EN LECTURAS DE SENSORES.....	31
4.5.2. COLA FIFO.....	32
4.5.2.1. PUT().....	32
4.5.2.2. GET().....	32
5. VALIDACIONES.....	34
6. PÁGINA WEB.....	35
7. CONSIDERACIONES FINALES.....	37
8. ANEXO.....	37
8.1 RECURSOS Y CÓDIGO FUENTE.....	37
8.2 BIBLIOGRAFÍA.....	38

Índice de figuras

Figura 1: Curva de saturación de la hemoglobina, en función de la presión parcial del oxígeno.....	10
Figura 2: Max30100.....	10
Figura 3: Max30102.....	10
Figura 4: Max30102. Leyendas de pinout en parte trasera.....	10
Figura 5: Poros y glándulas sudoríparas en la piel.....	12
Figura 6: GSR con guantes para los dedos.....	12
Figura 7: Dispositivos conectados al bus I2C.....	13



Figura 8: Proceso de la conversión A/D.....	14
Figura 9: ADS1115.....	15
Figura 10: Dispositivo ESP32 utilizado. Se indica la leyenda de pinout.....	16
Figura 11: Raspberry Pi Zero 2 W. Se muestra layout de pinout.....	17
Figura 12: Radxa Zero.....	19
Figura 13: Radxa Zero pinout.....	19
Figura 14: Raspberry Pi 400.....	21
Figura 15: Raspberry Pi 400 por dentro.....	21
Figura 16: VSCode y desarrollo en remoto por SSH.....	23
Figura 17: Arquitectura cliente-servidor propuesta.....	23
Figura 18: Diagrama de secuencia.....	25
Figura 19: Sensores detectados en bus I2C.....	26
Figura 20: PostgreSQL - Certificado servidor.....	27
Figura 21: Lectura secuencial y llenado de buffer de luz roja e infrarroja.....	29
Figura 22: Llamada a lectura secuencial de led rojo e infrarrojo.....	30
Figura 23: Execute_batch.....	31
Figura 24: Cifrado contraseña base de datos.....	31
Figura 25: Implementación ADC ADS1115.....	31
Figura 26: Obtener ID.....	33
Figura 27: generar_buffer(). Concurrencia de pool y llenado de cola FIFO.....	33
Figura 28: Cola FIFO. Sensores y BBDD.....	34
Figura 29: Llenar cola FIFO.....	34
Figura 30: Sacar elementos de la cola FIFO e insertarlos en la BBDD.....	35
Figura 31: Rutina principal.....	35
Figura 32: Carga y CPU en top.....	36
Figura 33: RAM libre.....	37
Figura 34: Rutinas página web y flask.....	37
Figura 35: Métricas usuarios en página web.....	38



Índice de tablas

Tabla 1: Características ESP32.....	17
Tabla 2: Características Raspberry Pi Zero 2 W.....	18
Tabla 3: Características Radxa Zero.....	20
Tabla 4: Diferencias de Raspberry Pi 400 sobre Raspberry Pi Zero 2 W.....	21
Tabla 5: Multiproceso.....	26
Tabla 6: Conexiones Raspberry PI y sensores.....	26



1. INTRODUCCIÓN

1.1. MOTIVACIÓN

Una carencia actual en el ámbito sanitario es la monitorización activa y remota de las constantes vitales de nuestros mayores y especialmente de las personas que tienen enfermedades o tienen una salud delicada. Desgraciadamente, la situación que hemos vivido durante los dos últimos años debido a la pandemia producida por el virus SARS-CoV-2 [1], nos ha hecho entender que nuestros sanitarios no tienen una visión del estado de estos colectivos para poder actuar lo antes posible ante cualquier situación de emergencia. Es por ello que esta monitorización activa podría servirnos además no sólo en situaciones de emergencia, sino que nos proporciona un mecanismo para ser proactivos para detectar arritmias u otras patologías en personas que aparentemente son sanas. También nos ayuda a detectar patrones extraños en los datos obtenidos y a aprender de los cambios en las constantes vitales entre personas catalogadas con algún tipo de patología.

La monitorización activa de las constantes vitales realizada de manera remota, implica también que los sanitarios podrían tener un histórico completo del estado de salud de los pacientes durante días, meses e incluso años. Esta información tan extensa es realmente valiosa ya que proporciona una cantidad de datos sobre las personas, que a día de hoy no se conoce. Además al realizarse de manera remota, el coste para los hospitales y esfuerzos humanos, se reduce al mantenimiento de los servidores de computación, el personal informático y los dispositivos electrónicos que utilizan los pacientes.

1.2. OBJETIVOS

Para que sea posible realizar la monitorización activa de los pacientes a gran escala, tiene que realizarse con dispositivos que proporcionen datos fiables, que dichos dispositivos sean de bajo coste, portables y que dispongan de una autonomía relativamente buena.

Durante los últimos años, ha habido avances muy importantes en la miniaturización de los dispositivos de computación. Existen dispositivos como los que vamos a detallar que tienen un coste muy bajo, un consumo eléctrico pequeño y una capacidad de proceso realmente interesante. Estas placas actuales distan mucho de las placas utilizadas hasta hace poco que eran muy limitadas en memoria y capacidad de computación. Las posibilidades de los últimos dispositivos son muy elevadas.

Además las redes móviles abarcan cada vez mayores coberturas. El futuro cercano se verá potenciado con la implementación de redes basadas en 5G [2] y protocolos de red como *IEEE 802.11ax* (WIFI 6) [6].

Una de las complicaciones más importantes es el envío de datos eficiente entre los diversos clientes portables a un servidor central. Veremos más adelante cómo podemos hacer provecho de las redes, así como de los mecanismos propuestos para tener una comunicación remota que sea lo más eficiente posible.



Finalmente indicar que este trabajo es parte de un proyecto que implica (a priori), el trabajo de tres personas.

- La primera de ellas trabajará con los sensores y los dispositivos a nivel físico para asegurar que los datos recogidos sean de manera óptima.
- La segunda, correspondiente al trabajo que nos atañe y detallaremos, se centra en la recogida y envío de datos de manera eficiente desde estos dispositivos a un servidor central.
- Una tercera persona, trabaaría a su vez con la interpretación y presentación adecuada de los datos recogidos.

1.3. ESTRUCTURA DEL DOCUMENTO

La presente memoria se estructura en siete capítulos que se comentan brevemente a continuación:

- El Capítulo 1 corresponde a la introducción realizada sobre el trabajo que nos atañe.
- En el Capítulo 2 se dota de contexto al proyecto mediante la revisión del estado del arte, en el que se describen y analizan las tecnologías y soluciones a nivel de investigación de sensores y dispositivos así como del *software* y herramientas tanto en la parte de cliente (dispositivos portables), como en la parte de servidor.
- En el Capítulo 3 se detalla el diseño del sistema, señalando y argumentando las decisiones tomadas.
- El Capítulo 4 comprende la parte de implementación, en la cual se verán las herramientas utilizadas a la hora de desarrollar los apartados vistos en el Capítulo 3.
- El Capítulo 5 se centra en mostrar las pruebas de validación realizadas, mostrando los resultados conseguidos a lo largo de la realización del proyecto.
- En el Capítulo 6 se muestran las conclusiones tomadas tras todo el ciclo de vida del proyecto, junto a las vías de trabajo futuras que quedan abiertas tras su finalización.
- El Capítulo 7 corresponde indica que se adjunta el código informático, además de referenciar el repositorio de la Universidad donde también se aloja el código del proyecto. También indica las referencias Bibliográficas.

2. ESTADO DEL ARTE

La siguiente definición del estado del arte pretende introducir los dispositivos utilizados, su finalidad y funcionamiento. Se ha buscado que los dispositivos escogidos sean los más óptimos para el proyecto que nos atañe, manteniendo en todo momento el coste bajo para que el diseño propuesto sea viable a gran escala. Adicionalmente se describirán los protocolos utilizados y la información conceptual necesaria.

2.1. SENORES

Definimos un sensor [4] como todo aquello que tiene una propiedad sensible a una magnitud del medio, y al variar esta magnitud también varía con cierta intensidad la

propiedad, manifestando la presencia de dicha magnitud y por tanto, también su medida. Un sensor en la industria es un objeto que puede variar una propiedad ante magnitudes físicas o químicas y transformarlas en variables eléctricas. Estas magnitudes pueden ser por ejemplo la intensidad lumínica, temperatura, distancia, aceleración, inclinación, presión, desplazamiento, fuerza, torsión, humedad, movimiento o nivel de pH.

2.1.1. PULSIOXIMETRÍA

La pulsioximetría [5] es un método no invasivo, que permite medir el porcentaje de saturación de oxígeno de la hemoglobina (SaO_2) en sangre de un paciente utilizando un circuito de fotoeléctricos. Para esto se emplea un pulsioxímetro, que es un dispositivo que integra los emisores de luz y el sensor que mide la cantidad de luz reflejada por el dedo del paciente. La luz detectada por el sensor varía de acuerdo a la concentración de oxígeno en la sangre, dado que la sangre oxigenada absorbe mayor cantidad de luz infrarroja y la sangre poco oxigenada absorbe mayor luz roja. Es por ello que los pulsioxímetros detallados a continuación miden tanto la luz roja como la luz infrarroja.

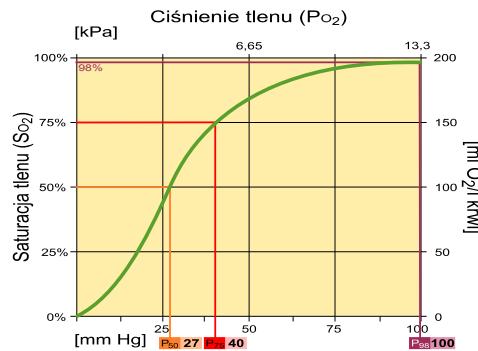


Figura 1: Curva de saturación de la hemoglobina, en función de la presión parcial del oxígeno. [6]

2.1.1.1. MAX30100 y MAX30102

Tal y como se indica en las hojas de datos del fabricante [7] y [8], ambos dispositivos son una solución de pulsioximetría y monitor de frecuencia cardíaca (FC) integrada. Combinan dos LEDs, un fotodetector, óptica optimizada y procesamiento de señal analógica de bajo ruido para detectar las señales de pulsioximetría y frecuencia cardíaca. Ambos dispositivos trabajan con fuentes de alimentación entre 1.8V y 3.3V



Figura 2: Max30100 [9]

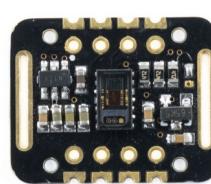


Figura 3: Max30102 [10]



Figura 4: Max30102. Leyendas de pinout en parte trasera [10]

Durante la realización del proyecto se ha trabajado con ambos dispositivos, eligiendo finalmente el dispositivo *Max30102*. Por un lado el dispositivo *Max30100* salió al mercado en 2014 y aunque se puede encontrar a la venta, está descatalogado. Por otro lado, tampoco existen diferencias de precio entre ambos dispositivos, pudiendo encontrarse ambos en el mercado por precios inferiores a los 2€. Además, de



mantenerse el precio, el dispositivo *Max30102* proporciona ventajas muy interesantes [11]:

- FIFO: el *Max30102* tiene un banco de memoria de 32 muestras, lo que significa que puede almacenar hasta 32 muestras de SpO₂ y frecuencia cardíaca. Mientras que el *Max30100*, sólo pueden guardar 16 muestras.
- Resolución *ADC*: el *Max30102* es capaz de leer hasta 18 bits o valores hasta 262.144. Mientras que el *Max30100* sólo tiene una resolución *ADC* de 16 bits. Por eso, el *Max30102* puede detectar movimientos extremadamente pequeños.
- Ancho de pulso LED: el *Max30102* tiene un ancho de pulso LED más estrecho que el *Max30100*, lo que resulta en un menor consumo de energía.

El *Max30102* es muy interesante también porque se puede programar para generar una interrupción [12]. Ello permite al microcontrolador anfitrión realizar otras tareas mientras el sensor recopila los datos. La interrupción se puede habilitar para 5 fuentes diferentes:

- *Power Ready*: se activa al encender o después de una caída de tensión.
- *New Data Ready*: se activa después de recopilar cada muestra de datos de SpO₂ y FC.
- *Ambient Light Cancellation*: se dispara cuando la función de cancelación de luz ambiental del fotodiodo SpO₂/HR alcanza su límite máximo, afectando a la salida del ADC.
- *Temperature Ready*: se activa cuando finaliza una conversión de temperatura interna de la matriz.
- *FIFO Almost Full*: se activa cuando la cola *FIFO* está llena y los datos futuros van a perderse.

La línea *INT* es de drenaje abierto, por lo que la resistencia integrada la eleva a ALTO. Cuando ocurre una interrupción, el pin *INT* pasa a BAJO y permanece BAJO hasta que se elimina la interrupción.

Durante las pruebas realizadas con ambos dispositivos, se han encontrado mejoras importantes al trabajar con *Max30102*. La gestión de la línea *INT* nos ha sido especialmente útil. Además la librería *Python* utilizada es también mejor y el cambio de dispositivo *Max30100* por el *Max30102*, ha sido esencial para trabajar con una estrategia mucho más eficiente como comentaremos posteriormente.

2.1.1.2. GSR (GALVANIC SKIN RESPONSE – RESPUESTA GALVÁNICA DE LA PIEL)

Nuestra piel funciona como la interfaz principal entre el organismo y el medio ambiente. Junto con otros órganos, es responsable de procesos corporales como el sistema inmunológico, la termorregulación y la exploración sensorio-motora. La piel, es también el órgano más grande que tenemos con cerca de tres millones de glándulas sudoríparas. La densidad del sudor producido por las glándulas varía a lo largo del cuerpo, siendo mayor en las palmas de las manos y los dedos, además de las suelas de los pies.

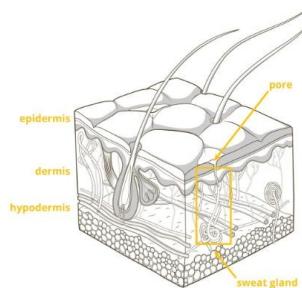


Figura 5: Poros y glándulas sudoríparas en la piel [13]

Uno de los sensores que integrará la solución final [13] implicará el estudio de la respuesta galvánica de la piel. Como hemos detallado en la introducción, el presente trabajo, es la segunda parte de un total de 3 trabajos. En el momento de realización de esta parte del proyecto, no se dispone todavía de la solución de código de la primera parte. Tampoco se dispone del dispositivo *hardware* y la crisis actual de componentes electrónicos, hace que los tiempos de entrega de los mismos sean largos. No obstante, las medidas realizadas con este dispositivo son analógicas y es necesario un convertidor *ADC* (detallado a continuación en el próximo apartado) para poder interpretar los datos con los dispositivos portables por los pacientes (ya que no disponen de convertidor *ADC* integrado). Con este código, demostramos la viabilidad del estudio a través de la implantación mediante multiproceso, y que mostraremos en el capítulo de implantación mediante valores aleatorios proporcionados por el *ADC*. Con ello la parte de código del *GSR*, debería ser ya prácticamente inmediata.

Los cambios en los patrones de sudor, recogidos a través de los guantes de los dedos, nos dan una métrica adicional interesante de cara a la monitorización activa de las constantes vitales de los pacientes. En el siguiente enlace [14] se proporciona la hoja de datos del dispositivo *GSR*, así como el código fuente de la librería Python y una posible implantación.

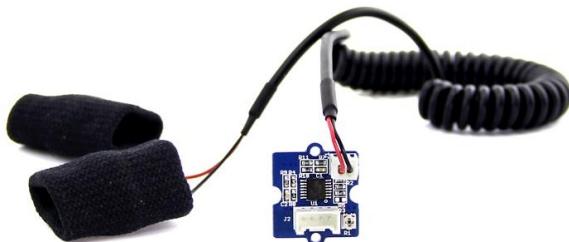


Figura 6: GSR con guantes para los dedos [14]

El *pinout* implica entrada de 5V, GND, NC y A0. El código del *ADC* que detallaremos, se implanta sobre esta entrada A0.

2.2. PUERTO Y PROTOCOLO I2C

I2C es un puerto y protocolo de comunicación serial que define la trama de datos y las conexiones físicas para transferir bits entre 2 dispositivos digitales. El puerto incluye dos cables de comunicación, *SDA* y *SCL*. Además el protocolo permite conectar hasta 127 dispositivos esclavos con esas dos líneas, con hasta velocidades de 100, 400 y 1000 kbits/s.



El protocolo *I2C* es uno de los más utilizados para comunicarse con sensores digitales, ya que a diferencia del puerto Serial, su arquitectura permite tener una confirmación de los datos recibidos, dentro de la misma trama, entre otras ventajas.

La conexión de tantos dispositivos al mismo bus, es una de las principales ventajas. Además si comparamos a *I2C* con otro protocolo serial, como *Serial TTL*, este incluye más bits en su trama de comunicación que permite enviar mensajes más completos y detallados. Los mensajes que se envían mediante un puerto *I2C*, incluye además del *byte* de información, una dirección tanto del registro como del sensor. Para la información que se envía siempre existe una confirmación de recepción por parte del dispositivo.

En una comunicación digital existen distintos dispositivos o elementos. En el caso de *I2C*, se diferencian dos elementos básicos (un maestro y un esclavo). La conexión típica entre estos elementos se realiza mediante un bus que consiste de dos líneas llamadas *Serial Data (SDA)* y *Serial Clock (SCL)* [18]

- *SCL* es la línea de los pulsos de reloj que sincronizan el sistema.
- *SDA* es la línea por la que se mueven los datos entre los dispositivos.
- *GND* (Masa) común de la interconexión entre todos los dispositivos «enganchados» al bus.

Las líneas *SDA* y *SCL* son del tipo drenaje abierto, es decir, un estado similar al de colector abierto, pero asociadas a un transistor de efecto de campo (o *FET*). Se deben polarizar en estado alto (conectando a la alimentación por medio de resistores *pull-up*), lo que define una estructura de bus que permite conectar en paralelo múltiples entradas y salidas.

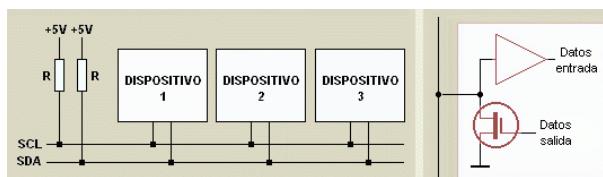


Figura 7: Dispositivos conectados al bus *I2C* [18]

El MAESTRO *I2C* se encarga de controlar al cable de reloj, por sus siglas en inglés llamada *SCL – Serial Clock*. Además el MAESTRO se encarga de iniciar y parar la comunicación. La información binaria serial se envía sólo por la línea o cable de datos seriales, en inglés se llama *SDA – Serial DAta*. Dos Maestros no pueden hacer uso de un mismo puerto *I2C*. Puede funcionar de dos maneras, como maestro-transmisor o maestro-receptor. Sus funciones principales son:

- Iniciar la comunicación – *S*
- Enviar 7 bits de dirección – *ADDR*
- Generar 1 bit de Lectura ó Escritura – *R/W*
- Enviar 8 bits de dirección de memoria
- Transmitir 8 bits de datos –
- Confirmar la recepción de datos – *ACK – ACKnowledged*
- Generar confirmación de No-recepción, *NACK – No-ACKnowledged*

- Finalizar la comunicación

El ESCLAVO *I2C*, generalmente suele ser un sensor. Este elemento suministra de la información de interés al MAESTRO. Puede actuar de dos formas: esclavo-transmisor ó esclavo-receptor. Un dispositivo *I2C* esclavo, no puede generar a la señal *SCL*. Sus funciones principales son:

- Enviar información en paquetes de 8 bits.
- Enviar confirmaciones de recepción, llamadas *ACK*.

2.3. ANALOG-TO-DIGITAL CONVERTER (ADC) - CONVERSOR ANALÓGICO DIGITAL

Un conversor de señal analógica a digital (*ADC*) [15] es un dispositivo electrónico capaz de convertir una señal analógica, ya sea de tensión o corriente, en una señal digital mediante un cuantificador y codificándose en muchos casos en un código binario en particular. Donde un código es la representación unívoca de los elementos, en este caso, cada valor numérico binario hace corresponder a un solo valor de tensión o corriente.

En la cuantificación de la señal se produce pérdida de la información que no puede ser recuperada en el proceso inverso, es decir, en la conversión de señal digital a analógica y esto es debido a que se truncan los valores entre 2 niveles de cuantificación, mientras mayor cantidad de bits mayor resolución y por lo tanto menor información perdida.

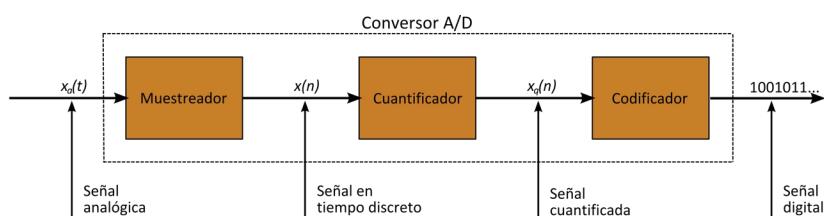


Figura 8: Proceso de la conversión A/D [15]

Definimos además el muestreo como el proceso de tomar muestras de la señal a intervalos periódicos. Es una modulación por amplitud de pulsos *PAM* (*pulse-amplitude modulation*) donde el desfase y la frecuencia de la señal quedan fijas y la amplitud es la que varía. Matemáticamente se expresa como:

$$x_s(t) = x(t) \sum_{k=0}^n P(t - kT_s) = \sum_{k=0}^n x(kT_s) P(t - kT_s) , \text{ donde:}$$

$x_s(t)$ es la señal muestreada

$\sum_{k=0}^n P(t - kT_s)$ es un tren de pulsos de período T_s

$x(t)$ es la señal a muestrear.



2.3.1 ADS1115

Los dispositivos de computación escogidos para que lleven los pacientes (y que detallaremos en el apartado posterior), no tienen convertidor de señal analógica a digital. Es por ello que hemos utilizado el convertidor externo que detallamos a continuación.

El dispositivo *ADS1115* [16] es un convertidor *ADC* de bajo coste (se puede conseguir por unos 3€), que proporciona 16-bit de precisión con 860 muestras / segundo sobre *I2C* [17]. Proporciona por tanto una precisión muy elevada de manera económica. Es además muy pequeño y de bajo consumo, siendo compatible con la portabilidad deseada.

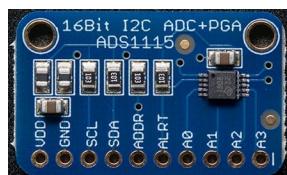


Figura 9: *ADS1115* [17]

El chip se puede configurar como 4 canales de entrada de un solo extremo o dos canales diferenciales (en nuestro caso utilizaremos sólo el canal *A0* además de las líneas *SDA* y *SCL* que irán sobre *I2C*). Como característica interesante adicional incluye un amplificador de ganancia programable, hasta x16, para ayudar a aumentar las señales simples/diferenciales más pequeñas a todo el rango. Este *ADC* puede funcionar con una potencia/lógica de 2 V a 5 V, puede medir una amplia gama de señales y es muy fácil de usar. Es un gran convertidor de 16 bits de propósito general.

Durante los capítulos de diseño e implantación, detallaremos las conexiones realizadas así como el código fuente utilizado.

2.4. DISPOSITIVOS DE COMPUTACIÓN PORTABLES Y DE BAJO COSTE

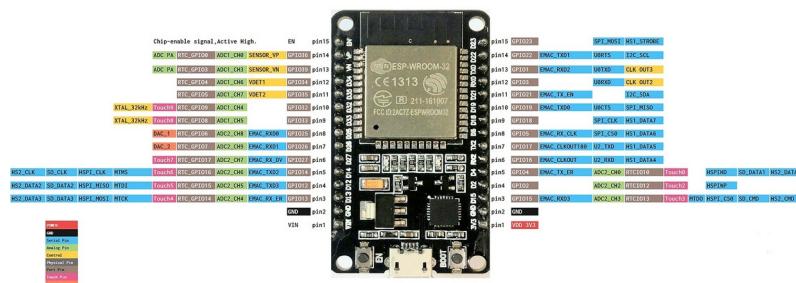
Durante la realización de este proyecto, se ha pretendido estudiar en profundidad varias de las alternativas más interesantes y se han utilizado diversos dispositivos hasta dar con la configuración correcta. Iremos detallando cada uno de ellos y en los apartados posteriores mencionaremos también las implantaciones realizadas.

2.4.1. ESP32

Tradicionalmente, se han utilizado las placas *Arduino Uno* [20] para tareas relacionadas con el manejo de todo tipo de sensores. Las placas que detallaremos a continuación son mucho más rápidas, más baratas (se pueden adquirir por 4\$) y tienen más memoria además de otras características como conexión *WiFi* integrada en el dispositivo. Actualmente, el único caso que podría hacer que nos decantemos por *Arduino* es por el gran soporte de la comunidad que ha realizado todo tipo de librerías sobre las mismas. Estas placas también son mejores para las entradas analógicas. *ESP32* [19] es la denominación de una familia de chips *SoC* de bajo costo (se pueden adquirir por 4\$), y bajo consumo de energía. Tienen además integrada tecnología *Wi-Fi* y *Bluetooth* de modo dual. El *ESP32* emplea un microprocesador *Tensilica Xtensa LX6* (en nuestro

caso utilizamos una variante de doble núcleo) e incluye interruptores de antena, balun de radiofrecuencia, amplificador de potencia, amplificador receptor de bajo ruido, filtros, y módulos de administración de energía. El *ESP32* fue creado y desarrollado por *Espressif Systems* y es fabricado por *TSMC* utilizando su proceso de 40 nm. Es un sucesor de otro *SoC*, el *ESP8266*.

Este dispositivo fué elegido durante las implantaciones inciales debido a su bajo coste y gran capacidad tanto de proceso como el *WiFi* integrado. Además dispone de *ADC* en el propio *SoC* y características especialmente interesantes como la criptografía acelerada por *hardware*. También ofrece la posibilidad de programar en *MicroPython*, como comentaremos brevemente en la implantación realizada.





A pesar de las ventajas obvias que tiene este dispositivo, nos hemos encontrado con varios inconvenientes que ha hecho que lo descartemos.

El primero de ellos es el bajo soporte de la comunidad. El mayor código fuente está realizado en C y en el momento que se probó esta placa, teníamos bastante código *Python* avanzado. Se utilizó una librería para dispositivos como el *Max30100* portada a *MicroPython* pero nos ocurrían varias cosas:

- *MicroPython* es bastante más limitado que *Python* y no soporta varias de las dependencias que estuvimos utilizando.
- Durante las pruebas realizadas, nos encontrábamos con problemas de memoria (detallado posteriormente).

Tanto en la memoria como en el repositorio de *software* utilizado, se adjunta el código fuente realizado.

2.4.2. RASPBERRY PI ZERO 2 W

Durante todas las pruebas, el dispositivo que se ha considerado ideal es el que se detalla en este apartado [21]. Dispone de la memoria y la capacidad de proceso adecuada, siendo muy superior al *ESP32*. Hay que entender que esta placa es ya un ordenador pequeño con un sistema operativo completo basado en *GNU/Linux*. Las posibilidades que ofrece no se limitan sólo por tanto al manejo de sensores sino que podría pensarse en utilizar la computación distribuida a través cada uno de los dispositivos de los pacientes.

El precio de este pequeño “ordenador” es de 15\$, siendo la relación calidad-precio muy interesante.

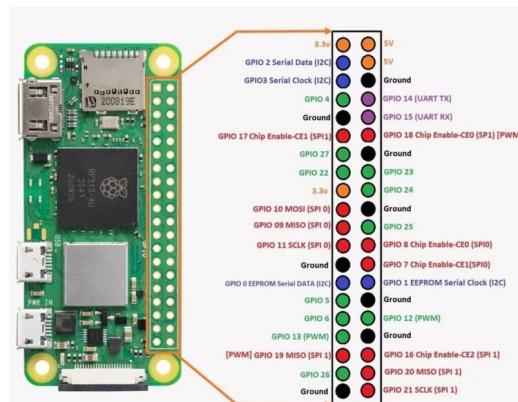


Figura 11: Raspberry Pi Zero 2 W. Se muestra layout de pinout [21]

Sus características más importantes son:



Specs	Detail
Procesador	Broadcom BCM2710A1, quad-core 64-bit SoC (Arm Cortex-A53 @ 1GHz)
Memoria	512MB LPDDR2
Red	2.4GHz IEEE 802.11b/g/n wireless LAN, Bluetooth 4.2, BLE.
Conectividad	1 × USB 2.0 con OTG. 40-pin I/O header. Slot MicroSD. Puerto mini HDMI port. Conector CSI-2 cámara.
Video	Interfaz HDMI. Video compuesto
Multimedia	H.264, MPEG-4 decode (1080p30). H.264 encode (1080p30). OpenGL ES 1.1, 2.0.
Potencia de entrada	5V DC 2.5A
Temperatura de trabajo	-20°C a +70°C
Form Factor	65mm × 30mm

Tabla 2: Características Raspberry Pi Zero 2 W

Con el desarrollo muy avanzado en esta placa, acabó dañada y no pude hacer que funcionara de manera correcta. La placa se reiniciaba constantemente. Se revisaron las soldaduras realizadas y finalmente se optó por buscar una placa alternativa. La crisis actual de chips hace que no haya placas basadas en *Raspberry Pi* en el mercado. Las fechas de envío son hacia el año 2023 y las placas que se consiguen duplican e incluso triplican su precio original. No he conseguido otra *Raspberry Pi Zero 2 W* pero el proyecto se ha podido terminar con una solución 100% compatible como detallaremos.

2.4.3. RADXA ZERO

Esta placa [22] tiene el mismo tamaño que la placa *Raspberry Pi Zero 2 W*. Se adquirió como posible alternativa a *Raspberry Pi Zero 2 W*, debido a su potencia muy superior. Además tiene *ADC* integrado, soporta criptografía por *hardware* y el dispositivo que se compró se eligió con módulo *eMMC*, siendo interesante ya que no es necesaria la tarjeta *SD*:

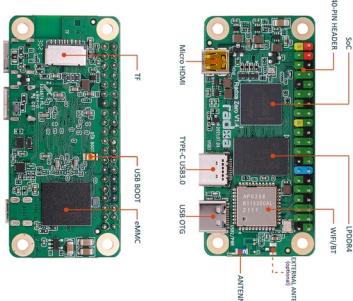


Figura 12: Radxa Zero [22]

Figura 13: Radxa Zero pinout [22]

La placa comprada con 4Gb de *RAM* y 16Gb de disco *eMMC* costó 65\$ con gastos de envío y aduanas. El coste es muy superior a *Raspberry Pi Zero 2 W*, pero también lo son las posibilidades que ofrece. Por otro lado, es interesante notar que la miniaturización nos proporcionará cada vez dispositivos más pequeños con mayor capacidad de proceso, por lo que la relación capacidad de proceso – precio puede ir variando. Las características principales de esta placa son las siguientes:

Specs	Detail
Procesador	Amlogic S905Y2 64bit quad core based Mini SBC. Quad Cortex-A53, frequency 1.8Ghz, 12nm
Memoria	4Gb LPDDR4 RAM. 64bit dual channel LPDDR4@3200Mb/s
GPU	Mali G31 MP2 GPU, supports OpenGL ES 1.1 /2.0 /3.1 /3.2, Vulkan 1.1, Open CL 1.1 1.2, 2.0 Full Profile
WiFi / Bluetooth	Wireless 802.11 ac wifi 2,4/5GHz. Bluetooth 5.0
Almacenamiento	EMMC 5.1 soldado. Slot MicroSD.
Cabezal de expansión 40-pin E/S	1 x UART. 2 x SPI bus. 2 x I2C bus. 1 x PCM/I2S. 1 x SPDIF. 1 x PWM. 1 x ADC. 6 x GPIO. 2 x 5V DC power in. 2 x 3.3V power pin
USB	USB 2.0 Type-C OTG x1. USB 3.0 Type-C HOST x1
HDMI	HDMI 2.1, 4K@60 HDR. HD codec H265/VP9 decode 4Kx2K@60
Otros	Crypto Engine

Tabla 3: Características Radxa Zero

Tras el problema ocurrido con *Raspberry Pi Zero 2 W*, se intentó aprovechar esta placa que ya disponíamos. Tras configurar el sistema operativo y el bus *I2C*, conectados los sensores, no los detectaba correctamente. Se intentó compilar también la librería Python del dispositivo *Max30100* hecha para *Raspberry Pi*, pero tampoco acababa de compilar correctamente.

De alguna manera, este dispositivo aunque es el más interesante en características (no requiere *ADC* externo, es muy potente con un consumo bajo, no necesita tarjeta *SD*)



externa y soporta criptografía por *hardware*), tiene una carencia importante dado que no tiene soporte de la comunidad. A día de hoy no hay librerías de sensores ni documentación. La información que podemos encontrar es exclusiva a la *Wiki* de *Radxa*.

Dado el estado avanzado del proyecto, ambas causísticas hicieron que se descartara esta placa.

2.4.4. RASPBERRY PI 400

Finalmente, tras el estado de incertidumbre al haber perdido la placa *Raspberry Pi Zero 2 W* y no poder conseguir a tiempo una placa compatible, dimos con un vendedor portugués que nos vendía una *Raspberry Pi 400*.

La *Raspberry Pi 400* [23] nos recuerda a aquellos ordenadores de la época de 8 bits, donde dentro de un teclado, viene el dispositivo de computación con todos sus puertos de entrada y salida. Su capacidad de proceso es muy similar al de *Raspberry Pi 4* [24], siendo ligeramente superior. Por lo tanto, teníamos una placa con exactamente el mismo sistema operativo que la placa *Raspberry Pi Zero 2 W* y que utilizaría las mismas librerías y versiones de *Python*.

Además el código fuente de las librerías de los sensores es exactamente el mismo. Con la placa compatible con el código fuente que habíamos desarrollado hasta el momento, continuamos trabajando en el proyecto sin seguir dando saltos entre alternativas. Podíamos avanzar a pesar del problema que habíamos tenido.



Figura 14: Raspberry Pi 400 [23]



Figura 15: Raspberry Pi 400 por dentro [23]

El *pinout* para conectar nuestros sensores *I2C* es también exactamente el mismo que el de *Raspberry Pi Zero 2 W*. Las características principales de esta placa son las ya detalladas en *Raspberry Pi Zero 2 W*, con las siguientes mejoras:

Specs	Detail
Procesador	Broadcom BCM2711B0, quad-core Cortex-A72
	Frecuencia de reloj 1,8Ghz
GPU	VideoCore VI 500 MHz
Memoria	4 GB LPDDR4-3200
Conectividad	Wi-Fi 802.11.b/g/n/ac. Bluetooth 5.0, BLE. Gigabit Ethernet

Tabla 4: Diferencias de Raspberry Pi 400 sobre Raspberry Pi Zero 2 W [23]



2.5. SOFTWARE

Detallamos a continuación muy brevemente todo el software utilizado tanto en los dispositivos portables detallados como en el servidor destino (donde como comentaremos, se alojará la base de datos y una página web que muestre los resultados)

2.5.1. LENGUAJES DE PROGRAMACIÓN

Durante el desarrollo del proyecto, se han utilizado los siguientes lenguajes:

2.5.1.1. PYTHON

Python es un lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma. Administrado por *Python Software Foundation*, posee una licencia de código abierto, denominada *Python Software Foundation License*.

Python es un lenguaje de programación muy popular en el ámbito científico y ampliamente utilizado en la Ingeniería de Sistemas. Además del lenguaje, se han utilizado múltiples librerías sobre el mismo que detallaremos durante la implantación.

2.5.1.2. MICROPYTHON

MicroPython [26] es una implementación del lenguaje de programación *Python* 3, escrita en C, optimizada para poder ejecutarse en un microcontrolador. *MicroPython* es un compilador completo del lenguaje *Python* a *bytecode* y un motor e intérprete en tiempo de ejecución del *bytecode*, que funciona en el *hardware* del microcontrolador. Al usuario se le presenta una línea de órdenes interactiva (el *REPL*) que soporta la ejecución inmediata de órdenes. Se incluye una selección de bibliotecas fundamentales de *Python*: *MicroPython* incluye módulos que permiten al programador el acceso al hardware en bajo nivel.

Esta implementación de *Python*, se utilizó con la placa ESP32.

2.5.1.3 HTML Y JSON

HTML (HyperText Markup Language) [27], hace referencia al lenguaje de marcado para la elaboración de páginas web. Es un estándar que sirve de referencia del software que conecta con la elaboración de páginas web en sus diferentes versiones, define una estructura básica y un código (denominado código *HTML*) para la definición de contenido de una página web, como texto, imágenes, videos, juegos, entre otros. Es un estándar a cargo del *World Wide Web Consortium (W3C)*, organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web, sobre todo en lo referente a su escritura e interpretación. *HTML* se considera el lenguaje web más importante siendo su invención crucial en la aparición, desarrollo y expansión de la *World Wide Web (WWW)*. Es el estándar que se ha impuesto en la visualización de páginas web y es el que todos los navegadores actuales han adoptado.

JSON (JavaScript Object Notation) [28], es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de *JavaScript*, aunque, debido a su amplia adopción como alternativa a *XML*, se considera

un formato independiente del lenguaje. Una de las supuestas ventajas de *JSON* sobre *XML* como formato de intercambio de datos es que resulta mucho más sencillo escribir un analizador sintáctico (*parser*) para él. En *JavaScript*, un texto *JSON* se puede analizar fácilmente, algo que ha sido fundamental para que haya sido aceptado por parte de la comunidad de desarrolladores.

2.5.2. POSTGRESQL

PostgreSQL [29], es un sistema de gestión de bases de datos relacional orientado a objetos y de código abierto, publicado bajo la licencia *PostgreSQL*, similar a la *BSD* o la *MIT*. Como muchos otros proyectos de código abierto, el desarrollo de *PostgreSQL* no es manejado por una empresa o persona, sino que es dirigido por una comunidad de desarrolladores que trabajan de forma desinteresada, altruista, libre o apoyados por organizaciones comerciales.

2.5.3. ENTORNO DE DESARROLLO

Tan sólo indicar muy brevemente que se puede trabajar con la placa *Raspberry Pi* desde un ordenador conectándose por *SSH* [35] y desarrollando en local como si fuera en remoto. El entorno de desarrollo basado en *Visual Studio Code* [34], tiene un plugin que permite el desarrollo en remoto por *SSH*. Si nos fijamos en la captura, veremos que estamos conectados por *SSH* a la placa *Raspberry Pi*, pudiendo trabajar con el editor desde nuestro *PC*:

main.py - sensores_raspberry [SSH: 192.168.53.241] - Visual Studio Code

Archivo Editar Selección Ver Ir Ejecutar Terminal Ayuda

EXPLORADOR main.py ADC.py obtener_valores.py max30102.py connect_bbdd.py cifrado_bbdd.py

SENSORES_RASPBERRY ...

_> __pycache__

_> Adafuit_ADS1x15

_> max30102

_> ADC.py

_> cifrado_bbdd.py

_> connect_bbdd.py

_> main.py

_> obtener_valores.py

main.py >

```
1 import connect_bbdd
2 # utilizamos time para crear buffer de 5 min antes de escribirlo en BBDD
3 import time
4 # necesitamos introducir timestamps en la BBDD para saber los momentos de las lecturas
5 from datetime import datetime, timezone
6 # importamos pytz porque datetime sólo trabaja con UTC y necesitamos hacer la conversión
7 import pytz
8 # intercambio rojo, infrarrojo y cálculo de valores
9 import obtener_valores
10 # Leemos ADC
11 import ADC
12 import psycopg2
13 import time
14 # Trabajaremos con multiproceso en el momento de generar el buffer
15 # Esto lo hacemos para leer paralelamente desde el ADC y el max30102
16 # Además utilizaremos multiproceso también para generar el buffer y escribir en la base de datos de forma concurrente.
17 import multiprocessing as mp
18 from multiprocessing import Process, Queue
19
20
21 ## Introducimos una rutina previa para obtener un ID único por placa, aprovechando que cada dirección MAC de la WLAN es única por dispositivo
22 ## Esto nos sirve como identificativo único de dispositivo y por tanto de usuario para poder hacer luego las Insert en la BBDD
23 import subprocess
24
25 def obtener_id():
26
27     string="ip addr show wlan0 | grep ether| awk '{print $2}'"
28     IDs=subprocess.getoutput(string)
29     return IDs
30
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS JUJITER
```

[09:34:19.747] Persisting server connection details to /home/oinalnz/.config/Code/User/globalStorage/es-vscode-remote.remote-ssh/vscodium-ssh-host-5c341dd6-6d9b74a76ca9c7733a29f0456fd8195d8f6dd0-0.84.0/data.json

[09:34:19.757] Starting forwarding server. localPort 43175 -> socksPort 34661 -> remotePort 42301

[09:34:19.757] [Forwarding 43175] Got connection 0

[09:34:19.759] Waiting for ssh tunnel to be ready

[09:34:19.759] [Forwarding 43175] Got connection 0

[09:34:19.761] Tunneled 42301 to local port 43175

[09:34:19.761] Resolved "ssh-remote+7b22886f3744e616d6522a22313922e3136382e35332e32343122c2275736572223a226f696e01747a227d" to "127.0.0.1:43175"

[09:34:19.771] -----

[09:34:19.793] [Forwarding server 43175] Got connection 1

[09:34:19.818] [Forwarding server 43175] Got connection 2

[09:34:21.146] [Forwarding server 43175] Got connection 3

SSH: 192.168.53.241 ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ WO ↻ Downloading IntelICode models: Downloading IntelICode models: 3/12834 KB (0%)

> ESQUEMA

> LÍNEA DE TIEMPO

Lin. 69 col. 21 Espacio: 4 UTF-8 LF Python 3.9.2 32-bit

Figura 16: VSCode y desarrollo en remoto por SSH

3. DISEÑO DE LA SOLUCIÓN PROPUESTA

3.1. ARQUITECTURA DE LA SOLUCIÓN PROPUESTA

Se propone la siguiente arquitectura:

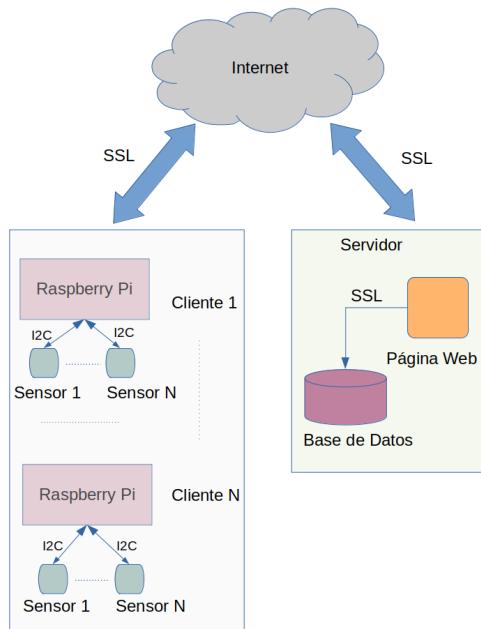


Figura 17: Arquitectura cliente-servidor pro-
puesta

En el diagrama anterior se pueden ver los N dispositivos cliente basados en *Raspberry Pi* y que serán portados por cada uno de los pacientes. Cada uno de estos dispositivos tendrá los sensores indicados conectados al bus *I2C*. Aunque hemos hablado de dos tipos de sensores (y un *ADC* externo), podríamos utilizar tantos sensores como soporte el bus *I2C*, la capacidad de proceso de cada *Raspberry Pi* y la capacidad de red para el envío de información.

La información recolectada por las mediciones de los sensores en cada *Raspberry Pi* de cada paciente, es enviada a través de Internet utilizando *SSL* y con un certificado autorizado, a una Base de Datos *PostgreSQL* que está en un servidor remoto.

Este servidor remoto, además de tener la Base de Datos de los datos de los sensores, también tiene una página Web que se alimenta de la Base de Datos para mostrar los datos recogidos por los sensores.

3.2 DIAGRAMA DE SECUENCIA

El funcionamiento del código realizado es el siguiente:

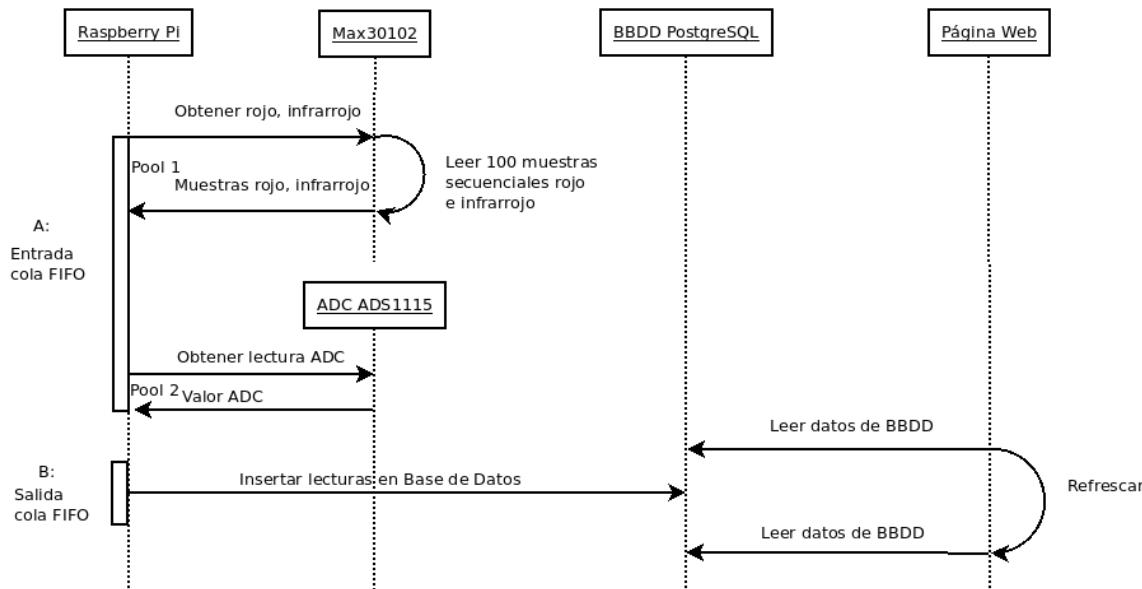


Figura 18: Diagrama de secuencia

Para poder entender el diagrama tenemos que hablar del multiproceso:

Proceso	Descripción
A:	Pool 1.- Por un lado se leen 100 muestras secuenciales de luz roja y otras 100 de luz infrarroja del dispositivo Max30102.
B:	Pool 2.- Por otro lado se leen valores del ADC

(Pool 1.-) y (Pool 2.-) se realizan de manera concurrente (dentro de A), aprovechando el multiproceso ya que el dispositivo utilizado tiene 4 CPUs (Raspberry Pi Zero 2 W tiene dos). Para ello, se utilizará un *pool* de multiproceso [30].

Utilizaremos la clase *Pool*, concepto que explicaremos porqué durante el apartado de implantación.

B: A su vez, concurrentemente con todo el mecanismo A, hay un proceso B que introduce los datos que se van leyendo a través del Pool en A, en la Base de Datos PostgreSQL. La salida de datos de A va a una cola FIFO de la cual se alimenta B [30]. Por lo que **A y B se realizan también de manera concurrente**. Explicaremos en el siguiente apartado este funcionamiento.

Tabla 5: Multiproceso

Finalmente, hay una página web que se nutre de los datos introducidos en la base de datos. Esta implantación, aunque no es objetivo de esta segunda parte del proyecto, se ha querido realizar para tener un portal en el que mostrar los datos introducidos. Cada vez que se refresca la página web, se vuelven a leer los datos introducidos en la Base de Datos.

4. IMPLANTACIÓN DE LA SOLUCIÓN

4.1 CONEXIONES E I2C

Para las conexiones de los sensores al bus *I2C*, se ha adquirido un pequeño adaptador con las leyendas de los pines. En los apartados del dispositivo *Max30102* como en el *ADS1115*, se han indicado ya las capturas de los *pinout* y lo único que queda es llevar el pin de cada leyenda, a los pines correspondientes en el dispositivo *Raspberry Pi*. Además recordamos que en cada bus *I2C*, se pueden conectar varios dispositivos.

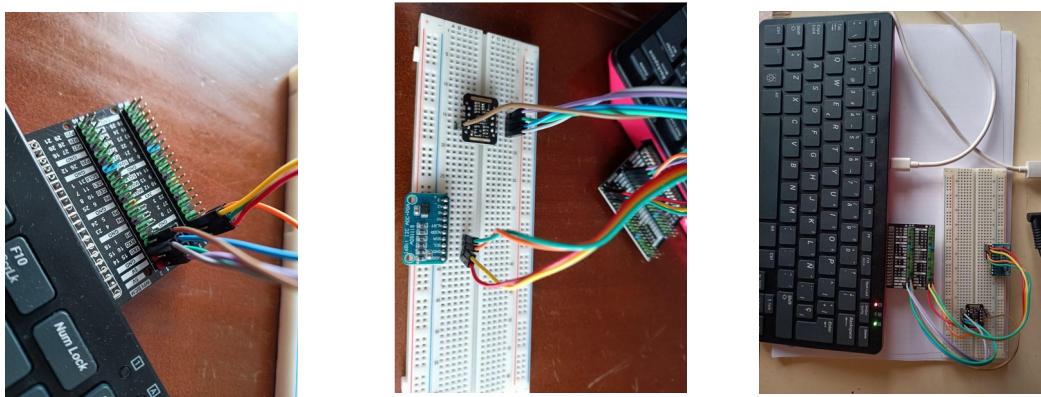


Tabla 6: Conexiones Raspberry Pi y sensores

Por otro lado, se configura la placa *Raspberry Pi* para habilitar el bus *I2C* tal y como indica la web del fabricante. Además, tras conectar los sensores, se buscan las direcciones del bus de ambos con *i2cdetect*.

```
oinatz@raspberrypi:~ $ i2cdetect -l
i2c-1  i2c          bcm2835 (i2c@7e804000)          I2C adapter
i2c-21 i2c          Broadcom STB :           I2C adapter
i2c-20 i2c          Broadcom STB :           I2C adapter
oinatz@raspberrypi:~ $ i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          - - - - - - - - - - - - - - - - - -
10:          - - - - - - - - - - - - - - - - - -
20:          - - - - - - - - - - - - - - - - - -
30:          - - - - - - - - - - - - - - - - - -
40:          - - - - - - - - 48 - - - - - -
50:          - - - - - - - - 57 - - - - - -
60:          - - - - - - - - - - - - - - - - - -
70:          - - - - - - - - - - - - - - - - - -
oinatz@raspberrypi:~ $
```

Figura 19: Sensores detectados en bus *I2C*

Los valores son hexadecimales. 0x57 es la dirección del bus *I2C* del dispositivo *Max30102* y 0x48 la del *ADS1115*. Ambos dispositivos están conectados al bus 1 (*i2c-1*). Podemos verlo también en la captura de las conexiones del cableado.



4.2 INSTALACIÓN Y MANEJO DE BASE DE DATOS

4.2.1. CONFIGURACIÓN RASPBERRY PI

En los dispositivos *Raspberry Pi*, hay que instalar una serie de librerías desde el usuario que nos conectamos. Con *apt-get install*, instalaremos también los paquetes *python3*, *python3-pip*, *python3-dev*. Instalaremos adicionalmente con *pip3 install* los paquetes *numpy*, *psycopg2-binary*, *Adafruit-ADS1x15*, *pandas*, *Rpi.GPIO*, *smbus*, *smbus2*, *urllib3*, *wheel*.

Finalmente en el */etc/hosts* de cada *Raspberry Pi*, introduciremos el nombre de la Base de Datos con la *IP* en la que responde. En nuestro caso:

192.168.53.108 rpsensors.com

4.2.2. CONFIGURACIÓN DE SERVIDOR DE BASE DE DATOS

Para que se pueda replicar la instalación hay que explicar cómo se ha configurado la Base de Datos ya que se ha pretendido que sea segura. Primero hay que instalar la Base de Datos *PostgreSQL* de manera estándar tal y como se indica en la documentación oficial [29]. Como comentábamos la Base de Datos se ha levantado con un certificado *SSL*. En el ejemplo lo generamos tal que:

```
openssl genrsa -des3 -out server.key 2048
chmod 400 server.key
openssl req -new -key server.key -days 36500 -out server.crt -x509 -subj
'/C=SP/ST=Bizkaia/L=Bilbao/O=UPV_EHU/CN=rpsensors.com/emailAddress=oaspiazu@gmail
.com'
```

Tanto el certificado (*server.crt*) como la clave privada del mismo (*server.key*), deben ser guardados en la carpeta *data* de la instalación de *PostgreSQL*. El campo más importante es el nombre de la base de datos *rpsensors.com*, nombre que utilizaremos posteriormente para realizar las conexiones a la misma de manera remota. En el ejemplo anterior se han utilizado datos personales que deberían ser reemplazados por los datos que se consideren oportunos en una instalación final.

Por otro lado, en el archivo *postgresql.conf*, configuraremos que el servidor de base de datos utilice el certificado creado, tal que:

```
# - SSL -
ssl = on
#ssl_ca_file =
#ssl_cert_file = '/etc/ssl/certs/ssl-cert-snakeoil.pem'
ssl_cert_file = '/var/lib/postgresql/data/server.crt'
#ssl_crl_file =
#ssl_key_file = '/etc/ssl/private/ssl-cert-snakeoil.key'
ssl_key_file = '/var/lib/postgresql/data/server.key'
```

Figura 20: PostgreSQL - Certificado servidor

En el archivo */etc/hosts* del servidor de Base de Datos, introducimos también las siguientes *IPs* y nombres:

192.168.53.108 bbdd_server
192.168.53.108 rpsensors.com



Finalmente, añadimos el usuario y las *IPs* autorizadas en el archivo *pg_hba.conf*:

```
# IPv4 local connections:  
host all all 192.168.2.108/32 md5  
host all all 192.168.2.212/32 md5  
....  
## Añadimos autenticación por ssl para el único usuario que hemos  
creado  
hostssl all sensors 192.168.128.211/32 md5  
clientcert=1  
hostssl all sensors 192.168.128.108/32 md5  
clientcert=1
```

Introduciremos tantas *IPs* como clientes autorizados. También introduciremos la *IP* del servidor. Para terminar, levantaremos la Base de Datos, crearemos la tabla de sensores con los campos que nos interesan y crearemos un usuario al que daremos privilegios:

```
su - postgres  
createuser sensors -P --interactive  
psql  
create database sensors;  
psql sensors  
create table sensores (usuario text, rojo text, infrarrojo text, pulso_spo2 text,  
fecha timestamp);  
GRANT ALL PRIVILEGES ON TABLE sensores TO sensors;
```

Reiniciamos la Base de Datos. Copiamos también el certificado *server.crt* a cada *Raspberry Pi*, a la carpeta destino que estimemos oportuna.

4.2.3. INSERTS EN BASE DE DATOS

Además de conectarnos de manera segura a la Base de Datos con los clientes *Raspberry Pi*, uno de los puntos más críticos y difíciles de resolver ha sido cómo realizar las *Inserts* de manera eficiente.

Como se puede comprobar en el código adjunto, se han realizado múltiples pruebas con mediciones de tiempos en las *Inserts* realizadas. Hay que entender que se pretende buscar una solución eficiente que consiga leer una gran cantidad de datos durante un tiempo constante. En este punto se han realizado dos mecanismos importantes de cara a obtener una solución adecuada.

4.2.3.1. READ_SEQUENTIAL()

Cada *Insert* lleva los datos de 100 medidas de luz roja y 100 medidas de luz infrarroja. Para ello invocamos el método *read_sequential()* de la librería *Max30102*.

```

def read_sequential(self, amount=100):
    """
    Función que lee el red led y el led ir 100 veces, dado el amount = 100
    Funciona como una función bloqueante
    """
    red_buf = []
    ir_buf = []
    for i in range(amount):
        while(GPIO.input(self.interrupt) == 1):
            # Sólo esperamos a la señal de interrupción, que indica que los datos están disponibles.
            pass

        red, ir = self.read_fifo()

        red_buf.append(red)
        ir_buf.append(ir)

    return red_buf, ir_buf
  
```

Figura 21: Lectura secuencial y llenado de buffer de luz roja e infrarroja

Gracias a la gestión de interrupciones, llenamos dos *arrays* con las 100 lecturas de cada tipo de led. Por lo tanto una única *Insert* lleva ya 100 medidas de cada led. Esta manera de realizar las *Inserts*, nos ayuda a incrementar muchísimo la resolución de los datos disponibles en la Base de Datos (100 únicas *Inserts*, llevarían ya por ejemplo 10.000 medidas de cada tipo de led)

Llamamos a este método de la siguiente manera:

```

import sys
import multiprocessing as mpr

### importamos las librerías de sensores. Creamos un paquete
sys.path.insert(0, '/home/oinatz/TFM_Bio/sensores_raspberry/max30102')
import max30102
import hrcalc
###


m = max30102.MAX30102()

# Se utilizan 100 muestras para calcular HR/SpO2 en cada loop

def coger_datos():
    #while True:
    red, ir = m.read_sequential()
    pulso_spo2 = hrcalc.calc_hr_and_spo2(ir, red)
  
```

Figura 22: Llamada a lectura secuencial de led rojo e infrarrojo

La línea adicional *pulso_spo2* llama a un método que estima el valor de pulso y *SpO₂* en sangre, en base a las 100 lecturas de led rojo y led infrarrojo. También devuelve valor *True / False* por cada tipo de led, en caso de que no esté colocado el dedo en el sensor.

4.2.3.2. PSYCOPG2.EXTRAS.EXECUTE_BATCH

Además de optimizar cómo se realizan las *Inserts*, tuvo que buscarse cuál era la manera más eficiente de realizarlas. Leyendo la documentación del capítulo de *extras* del driver de *PostgreSQL* para *Python* (*pyscopg2*), se encontró una anotación muy interesante en la misma [31] :

Fast execution helpers

The current implementation of executemany() is (using an extremely charitable understatement) not particularly performing. These functions can be used to speed up the repeated execution of a statement against a set of parameters. By reducing the number of server roundtrips the performance can be orders of magnitude better than using executemany().

```
psycopg2.extras.execute_batch(cur, sql, argslist, page_size=100)
```

Execute groups of statements in fewer server roundtrips.

Después de muchas pruebas, vimos que utilizando *execute_batch* se reducía drásticamente el tiempo de cada *Insert* realizada en la Base de Datos. Para ello, añadimos el *import* del módulo de *extras*, tal que:

```
import psycopg2
from psycopg2 import OperationalError, errorcodes, errors
import psycopg2.extras as extras
```

La parte de código que realiza la *Insert* con este método, es la siguiente:

```
postgreSQL_insert_Query = "INSERT INTO sensores (usuario, rojo, infrarrojo, pulso_spo2, sudoracion, fecha) VALUES (%s,%s,%s,%s,%s);"
valores=[(usuario, rojo, infrarrojo, pulso_spo2, sudoracion, fecha)]
psycopg2.extras.execute_batch(cursor, postgreSQL_insert_Query, valores)
connection.commit()
```

Figura 23: Execute_batch

4.2.3.3. CIFRADO CONTRASEÑA BASE DE DATOS

Se ha implementado una rutina para cifrar las contraseñas de la Base de Datos y no tener que enviarlas en texto plano (recordamos además que se ha implementado la conexión con *SSL* y que se requiere un certificado). También se limita la conexión a la Base de Datos por *IPs* autorizadas.

La rutina adicional es la siguiente:

```
from cryptography.fernet import Fernet
clave = b'RngMa8T0INjEafksaq2aafoZXEuwKI7wDe4c1F8AY='
cipher_suite = Fernet(clave)
texto_cifrado = b'gAAAAABiJ4zo9YDKL2YmvwJMqXRTa3GQZoTRCKxnEk3M7xNtTulKK-12qhzyP_LNzWqWHrv_BNarWbVpd4y4y5d3cgLxIdnQ=='
texto_descifrado = (cipher_suite.decrypt(texto_cifrado)).decode("utf-8") # la suite nos devuelve un literal byte y necesitamos un string
```

Figura 24: Cifrado contraseña base de datos

Como podemos comprobar se basa un *hash* binario a la clave de la Base de Datos para cifrarla con ese *hash*. Sólo se puede por tanto descifrar la contraseña conociendo el *hash*.

4.3. ADC ADS1115

Dadas las optimizaciones con las *Inserts*, se ha estimado necesario explicar cómo trabaja el código fuente con las lecturas del dispositivo *Max30102* en el apartado de manejo de datos (ya que estaba directamente relacionado).



Como hemos comentado, también se ha implantada la lectura del canal *A0* del *ADC ADS1115* con la librería proporcionada en el código. Esta implantación se ha realizado de la siguiente manera:

```
import time
# Importamos el módulo del ADC
import Adafruit_ADS1x15

def leer_adc():

    # Creamos una instancia ADS1115 ADC (16-bit).
    # Lo seteamos en el bus1, 0x48.
    adc = Adafruit_ADS1x15.ADS1115(address=0x48, busnum=1)

    # Elegimos una ganancia de 1 para leer voltajes de 0 a 4.09V.
    # Ver la tabla 3 del datasheet ADS1015/ADS1115 para otros valores.
    GAIN = 1

    # Sólo cogemos A0. No necesitamos más para el GSR.
    values = [0]*1
    for i in range(1):
        # Dado el rango de 1, leemos sólo A0 con la ganancia especificada
        values[i] = adc.read_adc(i, gain=GAIN)

    # Guardamos los valores formateados
    valores_adc = '{0:.6}'.format(*values)
    return (valores_adc)
```

Figura 25: Implementación ADC ADS1115

Este código lo que hace es primero crear , una instancia de la clase *ADS1115* del módulo importado *Adafruit_ADS1x15*. Posteriormente y para una Ganancia de 1, se llama al método *read_adc* para el canal *A0*. Al no tener nada conectado a este pin, el *ADC* nos devuelve un valor ficticio. En caso de que estuviera el sensor *GSR*, obtendríamos los valores buscados.

4.4. OBTENER_ID()

Esta pequeña rutina, aunque sencilla, se procede oportuno explicarla. Para realizar las *Inserts* en la Base de Datos, hay que contemplar que proceden de un identificador único. Es decir, cada *Raspberry Pi* pertenece a un único usuario y hay que hacer que los datos de esa *Raspberry Pi*, queden en la Base de Datos con el identificador correspondiente.

Se ha adoptado la siguiente solución:

```
### Introducimos una rutina previa para obtener un ID único por placa, aprovechando que cada dirección MAC de la WLAN es única por dispositivo
### Esto nos sirve como identificativo único de dispositivo y por tanto de usuario para poder hacer luego las Insert en la BBDD
import subprocess

def obtener_id():

    string="ip addr show wlan0 | grep ether| awk '{print $2}'"
    ID=subprocess.getoutput(string)
    return(ID)
```

Figura 26: Obtener ID

Se ha pensado en hacerlo así porque las *MACs* de red de cada dispositivo son únicas. Es decir, la *MAC* del dispositivo *WiFi* que tiene la placa *Raspberry Pi*, es única y exclusivamente el de ese dispositivo de esa *Raspberry Pi*. Entonces, si al hacer la *Insert* en Base de Datos, introducimos un campo con esta *MAC*, sabremos de qué *Raspberry Pi* proceden las métricas.

Además es interesante, porque los datos de *MAC* son de alguna manera anónimos. A pesar de enviar la información cifrada, no se envían datos de usuarios. Es una vez en el servidor, donde en una implementación final, podríamos tener una tabla adicional con la correspondencia *MAC – ID* real del usuario (*DNI* ó número de tarjeta sanitaria por ejemplo).



4.5. MULTIPROCESO

4.5.1. POOL EN LECTURAS DE SENSORES

Como hemos introducido en el diagrama de secuencia, la información del dispositivo *Max30102* y la del *ADC ADS1115* se recoge de manera concurrente utilizando la clase *Pool* [30]. La clase *Pool* nos permite ejecutar múltiples *jobs* por proceso, siendo diferente al uso de la clase *Process* en multiproceso que reservaría memoria en todos los procesos. La clase *Pool* tiene en cuenta el número de los procesos que trabajan en el *pool* y levanta los procesos. La clase *Pool* puede por tanto distribuir los trabajos y recolectar los resultados de los procesos levantados con una presencia mínima de procesos levantados.

Por otro lado utilizamos la función *apply_async()* para devolver los valores inmediatamente después de que la ejecución se completa. Este método **mantiene el orden de los resultados y soporta concurrencia**.

```
#Generamos el buffer y lo llenamos el buffer con los datos de los sensores
def generar_buffer(q):

    # Obtenemos el ID, único por dispositivo
    id=str(obtener_id())

    ### Obtenemos valores rojo / infrarrojo en un procesador dedicado
    # Dado que disponemos de varios procesadores, vamos a aprovechar el multiproceso
    # dedicando uno de ellos a las lecturas de POX y el otro al ADC. Con los datos de ambos, los introduciremos en una cola FIFO (Queue)

    while True:
        # Tenemos que usar pytz para sacar la fecha adecuada ya que datetime sólo trabaja con UTC.
        fecha = datetime.now(pytz.timezone("Europe/Madrid"))

        # Hay que crear la cola FIFO con 6 datos:
        # Usuario, rojo, infrarrojo, Pulso SP02, sudoracion, fecha
        # De momento introducimos sudoracion como valor ficticio según el valor ficticio obtenido con el ADC
        # El pulso y spo2 corresponden realmente a los valores de rojo e infrarrojo. Se hace media de 100 valores de cada rojo e infrarrojo en cada linea

        # Sirviéndonos del multiproceso, utilizamos un pool de 2 procesos para obtener valores concurrentemente del max30102 y el ADC.
        pool = mpr.Pool(processes=2)

        resultado_async_max30102 = pool.apply_async(obtener_valores.coger_datos).get()
        resultado_async_adc = pool.apply_async(ADC.leer_adc).get()

        # resultado_async_max30102[0] tiene 100 valores de sensor rojo
        # resultado_async_max30102[1] tiene 100 valores de sensor infrarrojo
        # resultado_async_max30102[2] tiene 4 valores: Pulso obtenido de las 100 lecturas, true/false, SP02 obtenido de las 100 lecturas y true/false.
        # Los valores de true/false son del pulso y spo2 para indicar si ha estado puesto el dedo y se han hecho lecturas.
        # resultado_async_adc tiene los valores reconocidos del ADC
        datos_sensores = [id , resultado_async_max30102[0], resultado_async_max30102[1], resultado_async_max30102[2], resultado_async_adc, fecha]
        # close() llama a destruir el pool y join() espera a los procesos trabajadores.
        pool.close()
        pool.join()

        ### Llenamos la cola FIFO con los datos de los sensores
        q.put(datos_sensores)
```

Figura 27: *generar_buffer()*. Concurrencia de pool y llenado de cola FIFO

En la captura podemos ver el *Pool* de 2 procesos. Para el primero de los procesos, utilizaremos la función *apply_async()* para trabajar concurrentemente con el dispositivo *max30102* y para el segundo con el dispositivo *ADC*.

El valor de *ID*, es el valor obtenido que se ha detallado en el apartado anterior. También se obtiene la fecha en formato *timestamp*. Todos los datos obtenidos se guardan en la lista de valores *datos_sensores*. Una vez creada la lista, se llama a los métodos *close()* y *join()* para destruir el *pool* y esperar a los procesos trabajadores. El motivo principal es buscar una gestión en memoria eficiente de los mismos.

4.5.2. COLA FIFO

Si nos fijamos en la captura anterior, la rutina *generar_buffer(q)*, recibe por parámetro la variable *q*. Esta variable es una cola *FIFO* [30]. Utilizaremos una cola *FIFO* como un canal de comunicación donde múltiples procesos trabajan con la misma. Nuestra cola trabaja según el siguiente diagrama:

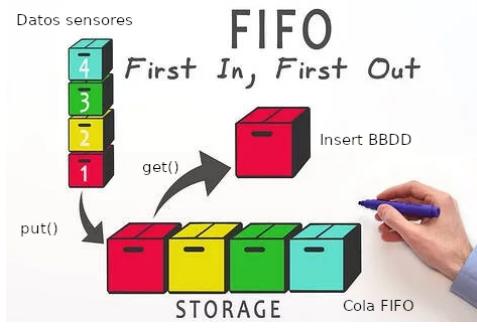


Figura 28: Cola FIFO. Sensores y BBDD

4.5.2.1. PUT()

En la captura anterior vemos como se invoca primero al método *put()*. Esto lo hacemos a través de:

```
### Llenamos la cola FIFO con los datos de los sensores
q.put(datos_sensores)
```

Figura 29: Llenar cola FIFO

Dentro de la rutina *generar_buffer(q)*. Cada línea formateada que lleva los datos de los sensores, el *ID* y el *timestamp* se introduce en la cola FIFO que se irá llenando de las N líneas por cada línea generada. Es interesante recordar que cada una de estas líneas lleva además 100 lecturas de led rojo y 100 lecturas de led infrarrojo, con el objetivo de conseguir una resolución muy elevada de datos.

4.5.2.2. GET()

Cuando invitamos al método *get()* de la cola, sacamos los elementos de la misma. Esta causística que implica sacar el elemento es importante ya que nos asegura que no vamos a duplicar elementos ni perder ninguno. Es decir, tenemos un objeto (la cola *FIFO*) y dos procesos que trabajan paralelamente con la misma, metiendo datos de sensores con *put()* y sacándolos con *get()* para insertarlos en la Base de Datos, de manera muy eficiente. A continuación se muestra la rutina completa que realiza las *Inserts* en la Base de Datos:

```
def insertar_sensores_bbdd_batch_queue(q):
    x=0
    ## Hacemos que se abra y cierre la conexión a la BBDD una única vez para todas las inserts.
    try:
        connection = psycopg2.connect(user="sensors",
                                       password=ifrado_bbdd.texto_descifrado,
                                       host="rpssensors.com",
                                       port= 5432,
                                       database="sensors",
                                       sslmode= 'require',
                                       sslrootcert = '/home/pi/.certs/postgresql.crt')
        print("Lanzamos la query...")
        cursor = connection.cursor()
        #Recorremos el buffer linea a linea para insertarlas en la BBDD
        while True:
            # Sacamos cada una linea de la cola FIFO
            linea = q.get()
            # Por cada linea, sacamos ahora los elementos individuales
            usuario=str(linea[0])
            rojo=str(linea[1])
            infrarrojo=str(linea[2])
            pulso_spo2=str(linea[3])
            sudoracion=str(linea[4])
            fecha=str(linea[5])
            # Preparamos la query con la insert
            postgresQL_insert_Query = "INSERT INTO sensores (usuario, rojo, infrarrojo, pulso_spo2, sudoracion, fecha) VALUES (%s,%s,%s,%s,%s,%s);"
            # Insert con execute batch
            valores=[(usuario, rojo, infrarrojo, pulso_spo2, sudoracion, fecha)]
            psycopg2.extras.execute_batch(cursor, postgresQL_insert_Query, valores)
            connection.commit()
            x+=1
            if linea is None:
                break
    except (Exception, psycopg2.Error) as error:
        print("Error obteniendo datos de la tabla de PostgreSQL", error)
    finally:
        # Cerramos la conexión a la BBDD
        if connection:
            cursor.close()
            connection.close()
            print("Cerrada la conexión a la BBDD\n")
```

Figura 30: Sacar elementos de la cola FIFO e insertarlos en la BBDD



La rutina `insertar_sensores_bbdd_batch_queue(q)`, recibe por parámetro la cola *FIFO*. Como se puede observar, por cada línea que sacamos de la cola con `get()`, obtenemos cada uno de los parámetros individuales para preparar la *Insert* que realizaremos en la Base de Datos con el método `execute_batch` detallado anteriormente. Además si se detecta que la cola está vacía (*linea is None*), no se realiza el `get()`. Todo ello queda en un bucle infinito (*while True*), para que se invoque siempre.

Dado que no hemos detallado esta rutina hasta que hemos hablado del multiproceso, hay que indicar que evidentemente (como se puede observar en la captura anterior) hay que hacer también la conexión a la Base de Datos, en este caso a través del método `pyscopg2.connect` de la librería `psycopg2`. Aquí se pasan los parámetros de la Base de Datos que implican las credenciales, puerto y nombre de escucha de la Base de Datos remota, el certificado *SSL* requerido para la conexión. El *password* se envía además cifrado con el método detalladamente.

Realizadas las *Inserts*, cerramos también las conexiones para no dejarlas abiertas. En este punto podemos detallar la rutina principal del programa:

```
### Rutina principal
def main():
    q = Queue()

    #### Llenamos la cola FIFO
    llenar_buffer = Process(target=generar_buffer, args=(q,))
    #### Vaciado de la cola FIFO
    envio_bbdd = Process(target=connect_bbdd.insertar_sensores_bbdd_batch_queue, args=(q,))
    llenar_buffer.start()
    envio_bbdd.start()

    llenar_buffer.join()
    envio_bbdd.join()

## Ejecución
if __name__ == "__main__":
    main()
```

Figura 31: Rutina principal

Podemos observar que tras definir la cola, asignamos al llenado de la cola a la función `generar_buffer` que añade los datos a la misma con `put()` y que recibe como argumento la cola. Por otro lado, asignamos al vaciado de la cola a la función `insertar_sensores_bbdd_batch_queue` que recibe la cola como parámetro y saca los elementos con `get()` para insertarlos en la base de datos con las optimizaciones ya comentadas anteriormente.

Finalmente comentar que antes de toda esta implantación se han realizado diversas implementaciones generando un *buffer* en *RAM* e irlo parando para hacer las *Inserts*. Se hicieron muchas pruebas utilizando tiempos y tamaños de *buffer* diferente pero siempre perdíamos muestras. Además la escritura por red en la Base de Datos, era bloqueante y hacía que las lecturas de los sensores quedaran en espera. Todo ello hacía que se incrementara la carga y se saturaba la máquina. El uso de multiproceso con los dos casos comentados (*pool* y cola *FIFO*), ha sido crucial para dar con la solución óptima.



5. VALIDACIONES

Se han realizado múltiples validaciones dejando una *Raspberry Pi* corriendo toda la noche, leyendo datos del dispositivo *Max30102* y el *ADC ADS1115* y realizando las *Inserts* de manera concurrente en la Base de Datos.

Por ejemplo, en una de las pruebas desde la 1:18AM hasta las 8:10AM (412 minutos), se han encontrado **6123** *Inserts* en la Base de Datos. Teniendo en cuenta que cada *Insert* lleva 100 lecturas de led rojo y 100 lecturas de led infrarrojo, tenemos el equivalente a **612300** *Inserts* de cada tipo de led. Esto implica un equivalente de **1.486,165** *Inserts* de cada tipo de led y **24,769** *Inserts por segundo de cada tipo de led*.

Tenemos que tener en cuenta que estamos hablando de datos ya insertados en la base de datos remota (y no datos de lectura de sensores), estimando que todas las optimizaciones comentadas, nos proporcionan una resolución realmente interesante.

Después de todas estas horas, un *top* en la máquina, nos muestra:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3146	oinatz	20	0	121592	29520	3780	R	43,2	0,8	0:01.30	python
3848	oinatz	20	0	121568	35936	10196	S	0,7	0,9	2:22.71	python
1362	oinatz	20	0	209160	47832	29820	S	0,3	1,2	0:40.46	node
3120	oinatz	20	0	11372	3044	2452	R	0,3	0,1	0:00.12	top
1	root	20	0	33868	8648	6772	S	0,0	0,2	0:03.70	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.04	kthreadd
3	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	rcu_par_gp
8	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_wq
9	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_rude_
10	root	20	0	0	0	0	S	0,0	0,0	0:00.00	rcu_tasks_trace

Figura 32: Carga y CPU en top

Si nos fijamos, entre los dos procesos *Python* levantados se consume cerca de un 44% de una única *CPU*. Hemos estado monitorizando durante bastante tiempo los proceso y son los valores más elevados que encontramos. Esta máquina tiene 4 *CPUs* (la placa *Raspberry Pi Zero 2 W*, tiene 2 *CPUs*), por lo que la máquina está trabajando muy holgadamente.

Además, si observamos en la misma captura el *load average*, se encuentra en 1 (dadas las 4 *CPUs*, los procesos están trabajando holgadamente), quedando validado el funcionamiento del sistema. A nivel de memoria *RAM*, ocurre algo similar. La placa con todo el sistema operativo con un escritorio completo levantado, sigue teniendo 3,1Gb de *RAM* libres:

```
oinatz@raspberrypi:~ $  
oinatz@raspberrypi:~ $ free -m  
total        used         free        shared      buff/cache   available  
Mem:       3838          666        2544           12          628        3111  
Swap:        99           0           99  
oinatz@raspberrypi:~ $
```

Figura 33: RAM libre



Esta *RAM* puede optimizarse mucho más dejando el sistema base más limpio pero el funcionamiento ya es correcto sin tener que limpiar nada.

6. PÁGINA WEB

Adicionalmente al proyecto, se ha realizado una pequeña página web en *Flask* [32]. *Flask* es un *framework* minimalista escrito en *Python* que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código. Está basado en la especificación *WSGI* (*Web Server Gateway Interface*, especificación que describe cómo se comunica un servidor web con una aplicación), de *Werkzeug* y el motor de plantillas *Jinja2* y tiene una licencia *BSD*.

Tras importar el módulo de *flask* de *Python*, el código, además de implementar las rutinas de conexión a Base de Datos y cifrado de *passwords*, tiene las siguientes líneas propias de *Flask*:

```
app = Flask(__name__)

@app.get('/usuarios')
def get_usuarios():
    conn = crear_conexion()
    cur = conn.cursor(cursor_factory=extras.RealDictCursor)
    cur.execute("SELECT * FROM sensores")
    usuarios = cur.fetchall()
    cur.close()
    conn.close()
    return jsonify(usuarios)

@app.get('/usuarios/<usuario>')
def get_user(usuario):
    conn = crear_conexion()
    cur = conn.cursor(cursor_factory=extras.RealDictCursor)
    cur.execute("SELECT * FROM sensores where usuario = %s", (usuario,))
    usuario = cur.fetchone()
    cur.close()
    conn.close()

    if usuario is None:
        return jsonify({'message': 'Usuario no encontrado'}), 404

    return jsonify(usuario)

@app.get('/')
def home():
    return send_file('static/index.html')

if __name__ == '__main__':
    app.run(debug=True, port=3000)
```

Figura 34: Rutinas página web y flask

- Para el enlace /usuarios de la página web se devuelven todas las entradas de la tabla sensores de la Base de Datos y se muestran en formato *JSON* en la página.
- Lo mismo se realiza para el enlace /usuarios/<usuario> pero sólo se muestran los datos de un usuario en concreto.
- Para la / de la web, se muestra una página index.html.

Esta página *index.html* es muy sencilla con una imagen del departamento y dos botones, uno por cada enlace detallado:

```
<!DOCTYPE html>
<html>
<head>
<title>LECTURA SENSORES RASPBERRY</title>
<link rel="stylesheet" href="{{ url_for('static', filename='home.css') }}">

</head>
<body><h1>Biosignals-SW</h1>
<h2>GICI-UPV-EHU</h2>
<div style="text-align: center;"><a href="GICI.png"></a></div>
<p>Esta página web muestra a modo de concepto los valores recogidos de los dispositivos MAX30102 y ADC ADS1115 en una Raspberry Pi, que han sido enviados a un servidor PostgreSQL. Los valores se presentan en formato JSON.</p>

<div style="text-align: center;"><p align="center"><a href="/usuarios "><input type="button" value="Métricas totales de usuarios"></a></p></div>
<div style="text-align: center;"><p align="center"><a href="/usuarios/e4:5f:01:14:7d:1c"><input type="button" value="Métricas de un usuario en concreto"></a></p></div>

</body>
</html>
```

La página web vista desde un navegador es (en las pruebas escucha en localhost:3000):



Métricas totales de usuarios

Si entramos en el primer botón podemos ver los valores de los usuarios con las métricas descritas anteriormente:

Figura 35: Métricas usuarios en página web

Lo mismo ocurriría en el botón que muestra un usuario en concreto, mostrando las métricas de ese usuario (para nosotros coincidiría ya que sólo tenemos una *Raspberry Pi* y por tanto una única *MAC* como identificador de usuario).



7. CONSIDERACIONES FINALES

En el presente documento se ha demostrado que el proyecto diseñado y realizado es viable. A nivel de código y lógica de funcionamiento el dispositivo implantado funciona correctamente gracias a las optimizaciones comentadas al trabajar con la Base de Datos y especialmente el uso de multiproceso.

Además es interesante tener en cuenta que estamos trabajando con dispositivos de muy bajo coste, haciendo que sea muy interesante su implantación final.

De cara a un diseño final habría que realizar múltiples validaciones con múltiples dispositivos. Además falta la parte inicial de la primera parte de este proyecto que valida que las métricas recogidas sean las mejores posibles y también la parte final que implique un diseño de una web final que interprete de manera visual todos los datos introducidos en la base de datos para todos los usuarios.

Un posible diseño final implica también que hay que tener en cuenta múltiples consideraciones de seguridad como por ejemplo separar la página web de la máquina de la base de datos o crear una *VPN* [33] para securizar aún más las conexiones, pudiéndose conectar a la Base de Datos sólo los dispositivos autenticados en la *VPN*. Recordamos que una *VPN* nos permitiría que los dispositivos trabajen como si estuvieran dentro de una red privada utilizando la red pública.

Una consideración final sería cifrar las tarjetas *SD* de las *Raspberry Pi* ya que son fácilmente extraíbles. Podría ser muy interesante utilizar un dispositivo con *eMMC* soldado y que además esté cifrado. Además habría que securizar los sistemas operativos *GNU/Linux* de cada *Raspberry Pi*. Finalmente, parece importante implantar un sistema de monitorización y alertas tanto a nivel de funcionamiento de los dispositivos como a nivel de métricas irregulares recogidas de cada paciente.

8. ANEXO

8.1 RECURSOS Y CÓDIGO FUENTE

El código fuente de todo el proyecto, se encuentra en el repositorio privado del departamento: <https://github.com/GICI-UPV-EHU/Biosignals-SW> . En este repositorio se han subido adicionalmente las pruebas iniciales realizadas con la placa *esp32* y esta misma memoria del proyecto, además de las transparencias de la defensa.

Se adjunta también el código fuente en un archivo *.zip*, junto con esta memoria y las transparencias de la defensa.

8.2 BIBLIOGRAFÍA

- [1] WIKIPEDIA. *SARS-CoV-2*. Actualizado en Agosto de 2022. Disponible en:
<https://es.wikipedia.org/wiki/SARS-CoV-2>



[2] WIKIPEDIA. *Telefonía móvil 5G*. Actualizado en Mayo de 2022. Disponible en:
https://es.wikipedia.org/wiki/Telefon%C3%A3_m%C3%B3vil_5G

[3] WIKIPEDIA. *IEEE 802.11ax*. Actualizado en Junio de 2022. Disponible en:
https://es.wikipedia.org/wiki/IEEE_802.11ax

[4] WIKIPEDIA. *Sensor*. Actualizado en Agosto de 2022. Disponible en:
<https://es.wikipedia.org/wiki/Sensor>

[5] NAYLAMP MECHATRONICS. *Pulioxímetro Max30100*. Actualizado en 2021.
Disponible en:
<https://naylampmechatronics.com/biomedico/328-pulioximetro-max30100.html>

[6] WIKIPEDIA. Saturación de oxígeno. Actualizado en Agosto de 2021. Disponible en:
https://es.wikipedia.org/wiki/Saturaci%C3%B3n_de_ox%C3%ADgeno

[7] MAXIM INTEGRATED. *Max30100 datasheet*. Publicado en 2014. Disponible en:
<https://datasheets.maximintegrated.com/en/ds/MAX30100.pdf>

[8] MAXIM INTEGRATED. *Max30102 datasheet*. Publicado en 2018. Disponible en:
<https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf>

[9] DIDACTAS ELECTRÓNICAS. *Sensor de concentración de oxígeno y ritmo cardíaco Max30100 (figura 2 de la memoria)*. Disponible en:
<https://www.didacticaselectronicas.com/images/stories/virtuemart/product/GY-MAX30100.jpg>

[10] NAYLAMP MECHATRONICS. *Pulsioxímetro Max30102 (figuras 3 y 4 de la memoria)*. Disponible en:
https://media.naylampmechatronics.com/2154-superlarge_default/pulsioximetro-max30102.jpg
https://media.naylampmechatronics.com/2156-superlarge_default/pulsioximetro-max30102.jpg

[11] LAST MINUTE ENGINEERS. *Interfacing MAX30100 Pulse Oximeter and Heart Rate Sensor with Arduino. (Apartado Max30100 vs Max30102)*. Publicado en 2022.
Disponible en:
<https://lastminuteengineers.com/max30100-pulse-oximeter-heart-rate-sensor-arduino-tutorial/>

[12] LAST MINUTE ENGINEERS. *Interfacing MAX30102 Pulse Oximeter and Heart Rate Sensor with Arduino (Interrupts)*. Publicado en 2022. Disponible en:
<https://lastminuteengineers.com/max30102-pulse-oximeter-heart-rate-sensor-arduino-tutorial/>

[13] IMOTIONS. *Galvanic Skin Response (GSR): The Complete Pocket Guide*. Publicado en Febrero de 2020. Disponible en:
<https://imotions.com/blog/galvanic-skin-response/>



[14] SEEDSTUDIO. *Grove GSR Sensor*. Actualizado en 2021. Publicado en:
https://wiki.seeedstudio.com/Grove-GSR_Sensor/

[15] WIKIPEDIA. *Conversor de señal analógica a digital*. Actualizado en Abril de 2022. Disponible en:
https://es.wikipedia.org/wiki/Conversor_de_se%C3%B1al_anal%C3%B3gica_a_digital

[16] ADAFRUIT. *ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier - STEMMA QT / Qwiic*. Disponible en:
<https://www.adafruit.com/product/1085> y
<https://learn.adafruit.com/raspberry-pi-analog-to-digital-converters/ads1015-slash-ads1115>

[17] HETPRO-STORE. *I2C – Puerto, Introducción, trama y protocolo*. Publicado en Febrero de 2018. Disponible en:
<https://hetpro-store.com/TUTORIALES/i2c/>

[18] ROBOTS ARGENTINA. *Descripción y funcionamiento del bus I2C*. Publicado en Junio de 2019. Disponible en:
<https://robots-argentina.com.ar/didactica/descripcion-y-funcionamiento-del-bus-i2c/>

[19] ESPRESSIF. *Esp32*. Actualizado en 2022. Disponible en:
<https://www.espressif.com/en/products/socs/esp32>

[20] WIKIPEDIA. *Arduino Uno*. Actualizado en Julio de 2022. Disponible en:
https://es.wikipedia.org/wiki/Arduino_Uno

[21] RASPBERRY PI. *Raspberry Pi Zero 2 W*. Actualizado en 2022. Disponible en:
<https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/>

[22] RADXA. *Radxa Zero*. Actualizado en 2022. Disponible en:
<https://wiki.radxa.com/Zero>

[23] RASPBERRY PI. *Raspberry Pi 400*. Actualizado en 2022. Disponible en:
<https://www.raspberrypi.com/products/raspberry-pi-400/>

[24] RASPBERRY PI. *Raspberry Pi 4*. Actualizado en 2022. Disponible en:
<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

[25] WIKIPEDIA. *Python*. Actualizado en Junio de 2022. Disponible en:
<https://es.wikipedia.org/wiki/Python>

[26] WIKIPEDIA. *MicroPython*. Actualizado en Diciembre de 2021. Disponible en:
<https://es.wikipedia.org/wiki/MicroPython>

[27] WIKIPEDIA. *HTML*. Actualizado en Agosto de 2022. Disponible en:
<https://es.wikipedia.org/wiki/HTML>

[28] WIKIPEDIA. *JSON*. Actualizado en Julio de 2022. Disponible en:
<https://es.wikipedia.org/wiki/JSON>



[29] POSTGRESQL. *PostgreSQL*. Actualizado en Agosto de 2022. Disponible en:
<https://www.postgresql.org/>

[30] ALLA, Samhita. *How to Use the Multiprocessing Package in Python*. Publicado en Julio de 2021. Disponible en:
<https://towardsdatascience.com/how-to-use-the-multiprocessing-package-in-python3-a1c808415ec2>

[31] PSYCOPG2. *Psycopg 2.9.3 documentation*. Actualizado en 2021. Disponible en:
<https://www.psycopg.org/docs/extras.html>

[32] FLASK. *Flask web development, one drop at a time*. Actualizado en 2022. Disponible en:
<https://flask.palletsprojects.com/en/2.2.x/>

[33] WIKIPEDIA. *Red privada virtual*. Actualizado en Mayo de 2022. Disponible en:
https://es.wikipedia.org/wiki/Red_privada_virtual

[34] MICROSOFT. *Visual Studio Code*. Actualizado en Agosto de 2022. Disponible en:
<https://code.visualstudio.com/>

[35] WIKIPEDIA. *Secure Shell*. Actualizado en Mayo de 2022. Disponible en:
https://es.wikipedia.org/wiki/Secure_Shell