

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías de
Telecomunicación

Diseño de una estación de recarga remota de coche
eléctrico basada en Arduino®

Autor: Guillermo Roche Arcas

Tutor: Alfredo Pérez Vega-Leal

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2013



Proyecto Fin de Grado
Grado en Ingeniería de Tecnologías de Telecomunicación

Diseño de una estación de recarga remota de coche eléctrico basada en Arduino®

Autor:

Guillermo Roche Arcas

Tutor:

Alfredo Pérez Vega-Leal

Profesor Contratado Doctor

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Proyecto Fin de Grado: Diseño de una estación de recarga remota de coche eléctrico basada en Arduino®

Autor: Guillermo Roche Arcas

Tutor: Alfredo Pérez Vega-Leal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El secretario del Tribunal

Agradecimientos

En primer lugar, me gustaría agradecer a mis padres, Francisco y Juana, por el apoyo que me han dado durante todos estos cursos de la carrera, y durante toda mi vida, animándome a seguir incluso cuando tenía la cabeza en otro sitio y veía esta carrera como un túnel sin salida de cálculo, física, ondas y abstracciones que no me parecían que tuvieran que ver con las Telecomunicaciones. Sin ellos y sin su apoyo, yo no estaría hoy escribiendo estas líneas.

Agradecer a mi hermano Carlos, que, aún siendo mucho más joven que yo, me ha enseñado que, aún cuando las cosas parecen muy malas, no hay que perder la alegría.

Tampoco me quiero olvidar de aquellas personas que han hecho esta etapa de mi vida mucho más llevadera, destacando a Adrián Vázquez y María Lao.

Agradecer también a Alfredo, por adjudicarme este trabajo y por el tiempo y la dedicación que me ha dedicado a lo largo de todo el curso.

Por último, quisiera agradecer a muchos de los profesores de la titulación las ganas y el esfuerzo que ponen en sus clases para no sólo transmitir conocimientos, sino una forma de ver las cosas y su pasión por lo que hacen, porque al final, un ingeniero tiene una visión diferente del mundo.

Índice

| | |
|---|-----------|
| Agradecimientos | 7 |
| Índice | 9 |
| Índice de ilustraciones | 11 |
| Índice de Figuras | 14 |
| 1 Introducción al proyecto | 2 |
| 2 Estaciones de recarga y vehículos eléctricos | 3 |
| 2.1 Estaciones de carga | 3 |
| 2.1.1 Mecanismos de seguridad | 4 |
| 2.1.2 Estandarización..... | 4 |
| 2.1.2.1 Clasificación según SAE. | 4 |
| 2.1.2.2 Clasificación según IEC..... | 4 |
| 2.1.3 Tiempos de recarga, autonomía y coste..... | 5 |
| 2.2 Tipos de conectores | 6 |
| 2.3 CCS (Combined Charging System) | 8 |
| 2.3.1 Comunicación | 8 |
| 2.3.2 Autenticación..... | 9 |
| 3 Descripción del sistema | 10 |
| 3.1 Estación remota (Lado cliente) | 10 |
| 3.1.1 Componentes utilizados y diagrama de bloques | 10 |
| 3.1.1.1 Placa Arduino Mega | 10 |
| 3.1.1.2 Tarjeta de red GPRS..... | 10 |
| 3.1.1.3 Pantalla LCD y expansor de pines | 11 |
| 3.1.1.4 Lector de códigos RFID | 11 |
| 3.1.1.5 Etapa de potencia | 12 |
| 3.1.1.6 Relé | 12 |
| 3.1.1.7 Cuadro eléctrico..... | 13 |
| 3.1.1.8 Diagrama de bloques del cliente | 15 |
| 3.1.2 Programa en lenguaje Arduino | 16 |
| 3.1.2.1 Subrutina setup..... | 16 |
| 3.1.2.2 Procesamiento de mensajes provenientes de la tarjeta de red | 16 |
| 3.1.2.3 Inicialización | 18 |
| 3.1.2.4 Ciclo de recarga..... | 22 |
| 3.1.2.4.1 Espera de asentimiento del servidor y de código RFID..... | 22 |
| 3.1.2.4.2 Envío de código RFID al servidor y respuesta | 22 |
| 3.1.2.4.3 Recarga e informe al servidor | 23 |
| 3.1.2.5 Subrutina: Lectura del código RFID | 25 |
| 3.1.2.6 Subrutina: gestión del ciclo de recarga | 26 |
| 3.2 Programa en lenguaje Python (Lado servidor) | 28 |
| 3.2.1 Ejecución y parámetros en línea de comandos. | 28 |

| | | |
|------------|--|-----------|
| 3.2.2 | Librerías utilizadas en el script | 29 |
| 3.2.3 | Inicialización | 29 |
| 3.2.3.1 | Comprobación del número de parámetros | 29 |
| 3.2.3.2 | Parámetros del socket | 30 |
| 3.2.3.3 | Conexión a la base de datos..... | 30 |
| 3.2.3.4 | Espera de conexión entrante | 31 |
| 3.2.4 | Funcionamiento..... | 32 |
| 3.2.4.1 | Espera de código RFID leído | 32 |
| 3.2.4.2 | Recepción y procesamiento de código RFID..... | 32 |
| 3.2.4.3 | Seguimiento y cómputo del ciclo de recarga..... | 34 |
| 3.2.4.4 | Registro de la recarga en la BBDD | 35 |
| 3.2.4.5 | Notificación por correo electrónico..... | 36 |
| 3.2.4.5.1 | Recogida de datos y composición del cuerpo del mensaje | 36 |
| 3.2.4.5.2 | Composición del objeto tipo 'email' | 37 |
| 3.2.4.5.3 | Autenticación en servidor SMTP y envío de correo | 38 |
| 3.2.5 | Diagrama de flujo simplificado completo del servidor | 39 |
| 4 | Montaje del sistema..... | 40 |
| 4.1 | Software..... | 40 |
| 4.1.1 | Software en cliente (Arduino IDE)..... | 40 |
| 4.1.2 | Servidor | 42 |
| 4.1.3 | Base de datos..... | 42 |
| 4.2 | Hardware (estación remota) | 45 |
| 4.3.1 | Montaje del cuadro eléctrico y relé | 45 |
| 4.3.2 | Alimentación | 47 |
| 4.3.3 | Sistema microcontrolador Arduino | 48 |
| 4.3.3.1 | Display LCD y expansor de pines..... | 48 |
| 4.3.3.2 | Tarjeta de red | 48 |
| 4.3.3.3 | Relé y medidor de consumo | 48 |
| 4.3.3.4 | Sensor RFID..... | 48 |
| | Referencias | 49 |
| | ANEXO A: Fundamentos de redes GPRS | 50 |
| 1. | Estructura y elementos de la red GPRS | 50 |
| 2. | Transferencia de paquetes entre nodos de red (GSN) GPRS..... | 51 |
| | ANEXO B: Protocolo NAT y configuración del router | 53 |
| 1. | Breve introducción teórica a NAT y su funcionamiento..... | 53 |
| 2. | Relación con el proyecto | 54 |
| | ANEXO C: Código Python del servidor..... | 56 |
| | ANEXO D: Código Arduino del cliente | 59 |

ÍNDICE DE ILUSTRACIONES

| | |
|---|----|
| ILUSTRACIÓN 1 SEÑALIZACIÓN DE TRÁFICO DE ESTACIONES DE RECARGA EN CARRETERA | 3 |
| ILUSTRACIÓN 2 PINOUT DE UN COMBO 2..... | 8 |
| ILUSTRACIÓN 3-1 PLACA ARDUINO MEGA2560..... | 10 |
| ILUSTRACIÓN 3-2 TARJETA DE RED NEOWAY M590..... | 10 |
| ILUSTRACIÓN 3-3 IZQ.: EXPANSOR DE PINES PCF8574, DER: LCD 16x02 | 11 |
| ILUSTRACIÓN 3-4 LECTOR RFID RC522 | 11 |
| ILUSTRACIÓN 3-5 FUENTE DE ALIMENTACIÓN Y CONVERTIDOR REGULABLE..... | 12 |
| ILUSTRACIÓN 3-6 RELÉ DE UN SOLO CONTACTO | 12 |
| ILUSTRACIÓN 3-7 CUADRO ELÉCTRICO COMPLETO | 13 |
| ILUSTRACIÓN 3-8 ESQUEMA DE CONEXIÓN DEL CUADRO ELÉCTRICO..... | 14 |
| ILUSTRACIÓN 3-9 MEDIDOR DE CONSUMO..... | 14 |
| ILUSTRACIÓN 3-10 EJECUCIÓN DEL SCRIPT SIN ESPECIFICAR NÚMERO DE PUERTO..... | 28 |
| ILUSTRACIÓN 3-11 NÚMERO DE PUERTO FUERA DE RANGO..... | 28 |
| ILUSTRACIÓN 3-12 CONSOLA DEL SERVIDOR CUANDO SE INTENTA AUTENTICAR UN USUARIO REGISTRADO EN LA BBDD | 33 |
| ILUSTRACIÓN 4-1 ARDUINO IDE DISPONIBLE A TRAVÉS DE LA TIENDA DE MICROSOFT..... | 40 |
| ILUSTRACIÓN 4-2 ARDUINO IDE EN LA WEB DE ARDUINO | 40 |
| ILUSTRACIÓN 4-3 INCLUSIÓN DE LIBRERÍAS EXTERNAS EN ARDUINO | 41 |
| ILUSTRACIÓN 4-4 OPCIÓN DE COMPILACIÓN DEL PROYECTO ARDUINO | 41 |
| ILUSTRACIÓN 4-5 CONFIGURACIÓN DE PUERTO USB EN EL ARDUINO IDE | 41 |
| ILUSTRACIÓN 4-6 CARGA DE PROGRAMA EN TARJETA ARDUINO | 41 |
| ILUSTRACIÓN 4-7 DESCARGA DE PYTHON PARA WINDOWS | 42 |
| ILUSTRACIÓN 4-8 RECORRIDO HASTA EL DIRECTORIO RAÍZ DE PYTHON | 42 |
| ILUSTRACIÓN 4-9 DESCARGA DE MYSQL WORKBENCH..... | 43 |
| ILUSTRACIÓN 4-10 CREACIÓN DE BBDD EN MYSQL | 43 |
| ILUSTRACIÓN 4-11 ACCESO AL MENÚ DE CREACIÓN DE TABLAS | 43 |
| ILUSTRACIÓN 4-12 EJEMPLO DE CREACIÓN DE TABLA..... | 44 |
| ILUSTRACIÓN 4-13 SCRIPT DE CREACIÓN DE TABLA GENERADO AUTOMÁTICAMENTE | 44 |
| ILUSTRACIÓN 4-14 VISUALIZACIÓN DEL CONTENIDO DE LA TABLA | 44 |
| ILUSTRACIÓN 4-15 VISUALIZACIÓN DEL CONTENIDO DE LA TABLA E INSERCIÓN DE NUEVAS FILAS | 45 |
| ILUSTRACIÓN 4-16 CONEXIÓN ENTRE LA RED ELÉCTRICA Y EL INTERRUPTOR GENERAL | 45 |
| ILUSTRACIÓN 4-17 CONEXIÓN ENTRE INTERRUPTOR GENERAL Y DIFERENCIAL | 46 |
| ILUSTRACIÓN 4-18 CONEXIÓN COMPLETA DEL CUADRO ELÉCTRICO..... | 46 |
| ILUSTRACIÓN 4-19 MEDICIÓN DE LA TENSIÓN DE SALIDA DE LA FUENTE DE ALIMENTACIÓN | 47 |
| ILUSTRACIÓN 4-20 AJUSTE DE LA TENSIÓN DE SALIDA DEL TRANSFORMADOR A 5V | 47 |
| ILUSTRACIÓN 0-1 CONFIGURACIÓN DE RED EN LA CONSOLA DE WINDOWS | 54 |
| ILUSTRACIÓN 0-2 ACCESO A LA CONFIGURACIÓN DE REENVÍO DEL ROUTER | 54 |
| ILUSTRACIÓN 0-3 CONFIGURACIÓN DE LA TABLA NAT MANUAL/ESTÁTICA DEL ROUTER..... | 55 |
| ILUSTRACIÓN 0-4 CREACIÓN DE NUEVA FILA ESTÁTICA EN LA TABLA NAT DEL ROUTER..... | 55 |

ÍNDICE DE FIGURAS

| | |
|---|----|
| FIGURA 1 REPRESENTACIÓN SIMPLIFICADA DEL PROYECTO | 2 |
| FIGURA 2 DIAGRAMA DE BLOQUES DEL CLIENTE..... | 15 |
| FIGURA 3 DIAGRAMA DE FLUJO DE LA FUNCIÓN SERIALEVENT() | 17 |
| FIGURA 4 OPERACIÓN BÁSICA DE COMPROBACIÓN DE CONECTIVIDAD | 18 |
| FIGURA 5 CONFIGURACIÓN DEL APN..... | 19 |
| FIGURA 6 COMPROBACIÓN DE LA CONEXIÓN A LA RED GPRS..... | 20 |
| FIGURA 7 ARQUITECTURA DE RED EN EL LADO DEL SERVIDOR..... | 20 |
| FIGURA 8 ESTABLECIMIENTO DE CONEXIÓN CON EL SERVIDOR | 21 |
| FIGURA 9 TABLA RESUMEN DEL PROCESO DE INICIALIZACIÓN | 21 |
| FIGURA 10 DIAGRAMA DE FLUJO RESUMEN DEL PROCESO DE ESPERA DE CÓDIGO RFID | 22 |
| FIGURA 11 DIAGRAMA DE FLUJO DEL ENVÍO Y RESPUESTA DEL CÓDIGO RFID AL SERVIDOR | 23 |
| FIGURA 12 SEGUIMIENTO DE LA RECARGA DESDE EL FLUJO PRINCIPAL DE PROGRAMA | 24 |
| FIGURA 13 RESUMEN DE LA SUBROUTINA DE LECTURA DE RFID | 26 |
| FIGURA 14 DIAGRAMA DE FLUJO DE LA SUBROUTINA DE GESTIÓN DE RECARGA | 27 |
| FIGURA 15 ACCESO A LOS PARÁMETROS DE EJECUCIÓN DEL SCRIPT | 29 |
| FIGURA 16 DIAGRAMA DE FLUJO DE LA INICIALIZACIÓN DEL SERVIDOR | 31 |
| FIGURA 17 ESTRUCTURA GENÉRICA DE UNA SENTENCIA SQL DE BÚSQUEDA EN BBDD..... | 32 |
| FIGURA 18 SENTENCIA SQL DE BÚSQUEDA EN BBDD DE USUARIO | 32 |
| FIGURA 19 SECUENCIA DE ACCIONES EN UNA CONSULTA A LA BBDD | 33 |
| FIGURA 20 DIAGRAMA DE FLUJO DE LA AUTENTICACIÓN | 34 |
| FIGURA 21 DIAGRAMA DE FLUJO DEL PROCESO DE RECARGA | 35 |
| FIGURA 22 ESTRUCTURA GENÉRICA DE UNA SENTENCIA SQL TIPO INSERT | 35 |
| FIGURA 23 SENTENCIA SQL DE INSERCIÓN EN BBDD | 36 |
| FIGURA 24 CONVERSIÓN DE CADENA DE TEXTO A OBJETO 'EMAIL' | 38 |
| FIGURA 25 DIAGRAMA DE FLUJO COMPLETO DEL SERVIDOR..... | 39 |
| FIGURA 26 ESTRUCTURA GENÉRICA DE UNA RED GPRS | 50 |
| FIGURA 27 TUNNELLING EN GPRS..... | 51 |
| FIGURA 28 ENCAPSULACIÓN DE MENSAJES EN REDES GPRS | 52 |
| FIGURA 29 TORRES DE PROTOCOLOS EN LA FRONTERA DE GPRS..... | 52 |
| FIGURA 30 FUNCIONAMIENTO BÁSICO DE NAT | 53 |

1 INTRODUCCIÓN AL PROYECTO

El proyecto que se describe en este documento consta de un sistema integrado por tres elementos funcionales que juntos forman un sistema de recarga remoto básico distribuido.

- Por una parte, tenemos la entidad que en la arquitectura distribuida representa el lado cliente. Está formado por una **placa Arduino y una serie de periféricos** conectados a la placa según la interfaz que sea necesaria.
- En el lado servidor, tenemos un proceso (**un programa escrito en lenguaje Python**) que se ejecuta en un equipo remoto en cualquier lugar de la interred. Es, por tanto, requisito indispensable para el funcionamiento del sistema que la estación de recarga sepa la dirección IP y puerto del servidor, no así, al contrario.
- Adicionalmente, el servidor se apoyará en **una base de datos relacional basada en el software MySQL**. Ésta guarda datos acerca de los usuarios registrados en el sistema, amén de otra tabla en la que se registrarán todas las recargas hechas, junto con la energía consumida durante las mismas.

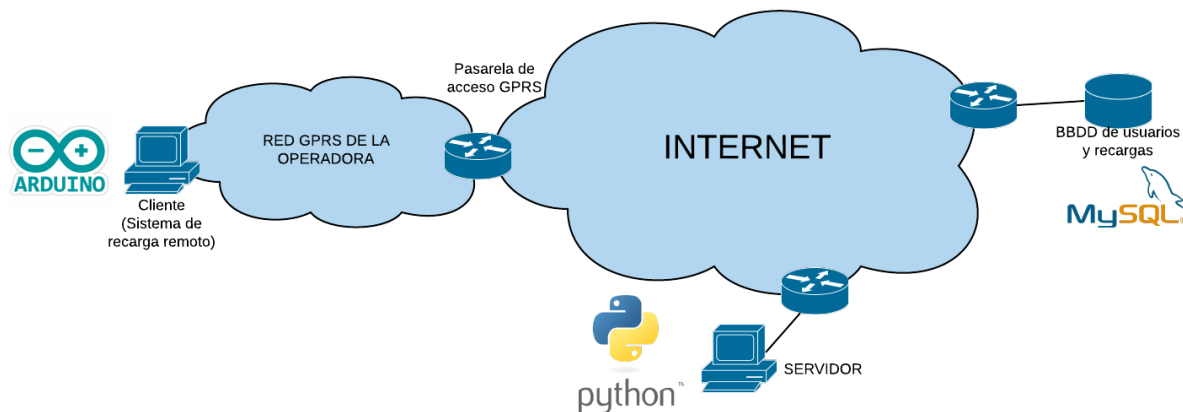


Figura 1 Representación simplificada del proyecto

Por pura generalidad se ha dibujado la parte correspondiente a la base de datos en un host independiente, sin embargo, lo común será que esté alojada en el mismo equipo que el proceso servidor.

También existe la posibilidad de emplear otra tecnología de red más actual en lugar de GPRS, sin embargo, para el poco caudal de información que se espera en esta aplicación las prestaciones de la red son más que suficientes.

El funcionamiento del sistema es, por tanto:

- El usuario se autentifica (ya se explicará cómo)
- El servidor realiza una consulta a la base de datos para comprobar si este usuario se encuentra en la base de datos.
- Dependiendo del resultado de la anterior consulta el servidor podrá dar orden a la estación de habilitar el suministro eléctrico al usuario que lo solicita (si encuentra coincidencia en la BBDD), o bien denegárselo, en caso contrario.
- La estación de recarga monitoriza el consumo eléctrico hasta que disminuye por debajo de un umbral mínimo (configurable desde el código), bien por desconexión de la carga, o bien porque la batería está llena.
- Cuando el proceso de recarga ha terminado se enviará un correo informativo al cliente.

2 ESTACIONES DE RECARGA Y VEHÍCULOS ELÉCTRICOS

Antes de comenzar la descripción detallada del proyecto del que trata este documento, pretendemos dar algunas nociones sobre el estado actual de esta tecnología.

2.1 Estaciones de carga

Una estación de carga de vehículo eléctrico es, definida de manera formal, un elemento dentro de una infraestructura que proporciona suministro eléctrico a vehículos que funcionan a base de baterías (total o parcialmente). Es frecuente en la literatura inglesa encontrarse las siglas ECS (*Electronic Charging Station*) y EVSE (*Electric Vehicle Supply Equipment*).



Ilustración 1 Señalización de tráfico de estaciones de recarga en carretera

Las estaciones de carga suelen tener a disposición de los usuarios varios conectores que cumplen diferentes estándares. Algunos de ellos con nombre propio son el CCS (Combined Charging System), CHAdeMO y el Tesla SuperCharger.

Los contextos básicos en los que puede darse una recarga son:

- En el ámbito privado del usuario. El vehículo es conectado a la red eléctrica cuando llega a casa, y el proceso dura típicamente toda la noche. No requiere que el usuario se autentique, tampoco requiere ningún proceso de medición de consumo.
 - En España la Ley 49/1960 del 21 de julio afirma lo siguiente: “La instalación de un punto de recarga de vehículos eléctricos para uso privado en el aparcamiento del edificio, siempre que éste se ubique en una plaza individual de garaje, **sólo requerirá la comunicación previa a la comunidad**. El coste de dicha instalación y el consumo de electricidad correspondiente **serán asumidos íntegramente por el o los interesados directos en la misma**”
 - Establece, resumidamente, que, aunque no se necesite permiso de la comunidad de propietarios para la instalación, no se podrá conectar el vehículo a la red eléctrica comunitaria ya que el coste del consumo eléctrico que cause debe correr a cuenta de las personas beneficiadas de su uso.
- Recarga en el parking de un establecimiento comercial. Como parte de su política para atraer clientes, existe la posibilidad de que las grandes superficies ofrezcan este servicio, bien sin coste adicional para el cliente, o bien aplicándole una tarificación.
- Recargas rápidas en estaciones públicas. Típicamente a una potencia superior a los 40kW. Están pensadas para los usuarios que estén realizando viajes relativamente largos. Estas estaciones son capaces de, en un tiempo inferior a media hora, proveer una autonomía cercana a los 100km.

- Cambios de batería. El proceso más rápido de todos. Suele realizarse en menos de 15 minutos.

2.1.1 Mecanismos de seguridad

Están pensados para detener el flujo eléctrico cuando el vehículo no se está cargando. Existen dos tipos de sensores de seguridad:

- Sensores de corriente. Vigilan la potencia consumida y cortan el suministro si la demanda de potencia no está en un rango predefinido.
- Otros elementos de seguridad, de implementación más compleja y con menos tiempo de reacción, proveen de una señal de “realimentación”, como se verá cuando se describa el conector tipo 1, también conocido como SAE J1772.

2.1.2 Estandarización

En esta sección introducimos dos organismos los cuales tienen su propia clasificación. Por una parte tenemos la **SAE** (*Society of Automotive Engineers*), de origen estadounidense, y por otro la **IEC** (*International Electrotechnical Commission*), de origen británico.

2.1.2.1 Clasificación según SAE.

Se basa en **las características eléctricas** del proceso. Define los ‘tipos’ como ‘niveles’ (*levels*)

| Nivel | Descripción |
|-------|--|
| 1 | Recarga usando la toma ordinaria doméstica estadounidense (120V AC). Una recarga completa, es decir, hasta el 100% de la capacidad de la batería tardará muchas horas, sin embargo, según el modelo con una noche podrá ser suficiente. Se adecúa a usuarios de trayectos cortos. |
| 2 | Recarga a 240V AC. Normalmente están situados en establecimientos comerciales o laborales. Está pensado para usuarios más frecuentes del vehículo o que requieren más flexibilidad. |
| 3 | A diferencia de los dos anteriores, el nivel tres se refiere a cargadores que usan tensión continua, pudiendo ser esta de hasta 500V. El organismo CHAdeMO está trabajando en una estandarización propia de las recargas de nivel 3. Típicamente estas recargas se realizan a 480V y una potencia de 62.5kW. El más famoso de este tipo es el ‘Tesla Supercharger’, que puede ofrecer casi 300km en 30 minutos de recarga. |

2.1.2.2 Clasificación según IEC

En esta clasificación, en lugar de ‘niveles’ hablamos de ‘modos’ (*modes*).

| Modo | Descripción |
|------|---|
| 1 | Carga lenta usando un enchufe doméstico. |
| 2 | Carga lenta usando un enchufe doméstico pero con algún tipo de protección implementada en el vehículo |
| 3 | Carga rápida o lenta usando algún tipo de control y protección multipin (por ejemplo el SAE J1772) |
| 4 | Carga rápida usando alguna tecnología especial como CHAdeMO |

2.1.3 Tiempos de recarga, autonomía y coste

De manera general, el tiempo de recarga puede calcularse mediante la expresión siguiente:

$$\text{Tiempo de recarga (h)} = \frac{\text{Capacidad de la batería (kWh)}}{\text{Potencia de carga (kW)}}$$

La capacidad de la batería de fabricantes como Nissan es de aproximadamente 20kWh, que proporciona una autonomía cercana a los 160km. Tesla, en su modelo S, cuenta con baterías de 40, 60 y 85kWh, teniendo la última una autonomía de casi 500km. En enero de 2018, sacaron al mercado modelos con 75 y 100kWh.

La capacidad de la batería de los coches híbridos suele ser mucho más reducida, alrededor de 5kWh, pero el motor de gasolina asegura la autonomía de un vehículo convencional.



Para una recarga normal (de aproximadamente 7.4kW) se puede usar la circuitería interior del vehículo, conectándolo directamente a la red eléctrica. Sin embargo, dependiendo del modelo, se puede llegar a los 43kW. Para velocidades más rápidas de carga se puede usar un cargador externo, que suministra corriente continua a 50kW o incluso 135kW en algunos modelos de Tesla.

| Tiempo de recarga para 100km | Modo | Potencia (kW) | Tensión (Voltios) | Corriente máxima (Amperios) |
|------------------------------|------------|---------------|-------------------|-----------------------------|
| 6-8h | Monofásico | 3.3 | 230 AC | 16 |
| 3-4h | Monofásico | 7.4 | 230 AC | 32 |
| 2-3h | Trifásico | 11 | 400 AC | 16 |
| 1-2h | Trifásico | 22 | 400 AC | 32 |
| 20-30 minutos | Trifásico | 43 | 400 AC | 63 |
| 20-30 minutos | DC | 50 | 400-500 DC | 100-125 |
| 10 minutos | DC | 120 | 400-500 DC | 300-350 |

En cuanto al coste, Tesla regala a los compradores de sus vehículos crédito canjeable por 400kWh sin coste alguno. Cuando este crédito se termina, se factura por cada kWh, estando el precio oscilando entre los 6 y los 26 centavos, dependiendo del lugar. Para los usuarios de vehículos de un fabricante diferente, está la red de cargadores del fabricante Blink, la cual aplica tarifaciones diferentes según se es socio o no (39-69c por kWh para socios y 49-79c por kWh para el resto).

2.2 Tipos de conectores

En vehículos de menor tamaño (ciclomotores o bicicletas eléctricas) es frecuente encontrarse un conector tipo Schuko. En automóviles está totalmente desaconsejado ya que la máxima corriente para la que está especificado es 10A

| Conector | Imagen | Descripción |
|--------------------------|---|---|
| Tipo 1 (SAE J1772) |  | <p>Es conector estándar en Japón y EE. UU. para recargas en corriente alterna.</p> <p>Trabaja en modo monofásico. Posee cinco pines, de los cuales tres son para el suministro eléctrico (fase, neutro y toma tierra) y dos pines de comunicación.</p> <p>Algunos fabricantes que implementan este conector son Toyota, Ford, Opel, Nissan, Mitsubishi, Peugeot...</p> <p>La máxima corriente para la cual está pensado es de 32A (7.4kW).</p> |
| Tipo 2 (Mennekes) |  | <p>El nombre de Mennekes no está normalizado, es debido a la empresa que lo desarrolló y lo comercializó primero. Está aprobado como un estándar en Europa.</p> <p>La diferencia a simple vista con el anterior es que este conector posee siete pines en lugar de cinco, ya que está pensado para poder llevar a cabo recargas en modo trifásico.</p> <p>En régimen monofásico ofrece menos corriente que el de tipo 1, únicamente 16A (3.5kW), pero en trifásica puede llegar a los 63 A (44kW).</p> <p>Algunos fabricantes que emplean este conector son Tesla, BMW, Renault, Porsche...</p> |

| | | |
|------------------------|--|---|
| <p>Tipo 3</p> |  | <p>Creado en 2010 por la <i>EV Plug Alliance</i>, actualmente se encuentra en desuso. Se comercializaron dos variantes:</p> <ul style="list-style-type: none"> • Modelo 3A: Para recargas en monofásica. Cuenta con pines de fase, neutro y toma a tierra. La corriente máxima especificada era de 16 A. • Modelo 3C: También soportaba recargas en trifásica. La intensidad máxima especificada era de 32 A. <p>La potencia máxima de este conector era de 22kW.</p> |
| <p>CHAdemo</p> |  | <p>Desarrollado por una asociación de empresas del sector en Japón, que incluía, entre otras, a Nissan, Mitsubishi y Toyota.</p> <p>La diferencia con respecto a los demás es que fue ideado desde el principio para recargas de corriente continua, permitiendo corrientes de hasta 125 A y 50kW de potencia.</p> <p>Fue equipado en algunos coches como el Nissan Leaf, el Nissan ENV200 y el Mitsubishi Outlander.</p> |
| <p>Combo 1/Combo 2</p> |  <p>High Power Charging TechnologyTM Designed by PHOENIX CONTACT</p> | <p>Este modelo es la adaptación al público europeo y norteamericano de las recargas en corriente continua. Como se intuye en el nombre, se trata de un conector Mennekes/tipo1 al cual se le han implementado dos terminales extra para el flujo de corriente continua. Gracias a este conector se puede recargar indistintamente mediante los modos 2, 3 y 4.</p> <p>La máxima potencia para la cual está diseñado es de 43kW, sin embargo, en algunos casos puede llegar a 100kW. En corriente continua está limitado a 50kW.</p> <p>Algunos fabricantes que lo incorporan son Audi y BMW</p> |

2.3 CCS (Combined Charging System)

Se trata de un entorno basado en diferentes estándares que describe los componentes necesarios para cargar de manera segura y fiable un vehículo de motor eléctrico.

Permite el uso de conectores tipo 1 y tipo 2 (basados, como explicamos en la tabla, en la localización geográfica), para la recarga en corriente alterna. Recientemente, se han desarrollado modelos mejorados, llamados Combo 1 y Combo 2 que permiten además la recarga usando corriente continua para acortar el tiempo necesario.

Los pines están claramente diferenciados en dos tipos, de control y de potencia.

- Los pines de control están orientados a la seguridad y señalización.
 - Proximity Pilot (PP): Definido como “señalización pre-inserción” Evita que el coche se ponga en marcha estando conectado.
 - Control Pilot (CP) Definido como “señalización post inserción”. Se usa para que la estación reciba información acerca del proceso.
- Los pines de potencia están diferenciados entre AC y DC.
 - Pines de fase (L). En el caso de tipo 1/combo 1, únicamente será uno (no soporta recarga trifásica), en el de tipo 2 serán tres.
 - Pin neutro (N).
 - Pines de recarga DC.

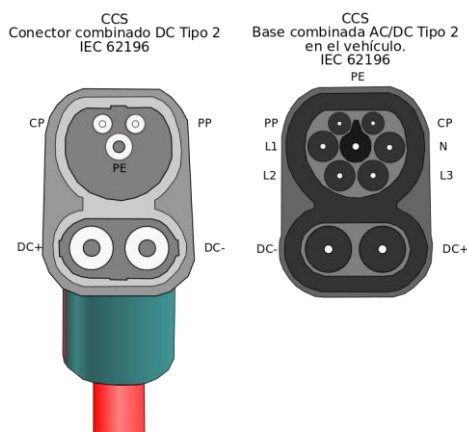


Ilustración 2 Pinout de un Combo 2

Es importante notar que, aunque con un conector AC (tipo 1/2) es posible recargar en un conector que soporta DC (Combo) no es así, al contrario.

2.3.1 Comunicación

La comunicación entre estación y vehículo obedece el mismo protocolo en todos los conectores, independientemente de la región.

- La señalización básica (referida como BS, *Basic Signalling*) se hace con una modulación PWM en el pin CP. Este canal se usa para comunicaciones relacionadas con la seguridad, por ejemplo, para cerciorarse de que el conector está encajado en el vehículo y listo para recibir corriente antes de dar corriente en los contactos (para evitar una descarga al manipular). Esta comunicación está regulada por la norma IEC 61851-1.
- La comunicación de alto nivel (HLC, *High Level Communication*) se lleva a cabo modulando una señal a alta frecuencia en el pin CP. Se usa para funcionalidad más compleja que la comunicación básica.

2.3.2 Autenticación

Para la identificación del usuario en el sistema se emplean dos estrategias fundamentales, con pequeñas variaciones dependiendo de la implementación.

- Con “plug and charge”, no se requiere interacción humana en el proceso de autenticación. El sistema es capaz de identificar el vehículo y el usuario con sólo conectar el vehículo. Al terminar la recarga se envía el cargo a la cuenta bancaria del usuario (o se le resta de algún tipo de saldo que contemple el sistema de la compañía)
- Si no está implementado lo anterior, el usuario deberá identificarse en el sistema mediante algún método (típicamente tarjeta RFID) o directamente pagar con tarjeta de crédito o débito.

3 DESCRIPCIÓN DEL SISTEMA

En esta sección se describirá de manera pormenorizada cómo se ha implementado el sistema descrito anteriormente.

3.1 Estación remota (Lado cliente)

Como ya adelantamos en la introducción se trata de un sistema basado en una placa Arduino, junto con diferentes periféricos y componentes.

3.1.1 Componentes utilizados y diagrama de bloques

3.1.1.1 Placa Arduino Mega

Es una placa basada en el microcontrolador **ATMega2560**. Funciona a una velocidad de reloj de 16MHz. Cuenta con 54 pines digitales configurables como entrada o salida (de los cuales 15 se pueden usar para una salida en PWM) y otros 16 para entrada analógica (para estos pines cuenta con un ADC de 10 bits generando una salida digital comprendida entre 0 y 1023).

Otras especificaciones que destacar son la existencia de tres pares (transmisor y receptor) de pines para **comunicación serie asíncrona (UART)**, pines dedicados a la **comunicación serie síncrona** (bus de datos bidireccional, SDA y señal de reloj SCK) y **salidas a 3.3V y a 5V** para alimentar periféricos, aunque en este proyecto no las usaremos (montaremos un sistema sencillo de alimentación)



Ilustración 3-1 Placa Arduino Mega2560

3.1.1.2 Tarjeta de red GPRS

Está basada en el chip M590 fabricado por Neoway. Hace de interfaz entre la placa Arduino y la red GPRS del operador de la tarjeta SIM que insertemos. Se comunica con el microcontrolador a través de comunicación serie asíncrona full-dúplex (lo que se conoce como UART). Para comunicarnos con ella tendremos que hacer uso de un conjunto de comandos que pueden consultarse en la web del fabricante. Permite, entre otras cosas, comunicaciones mediante TCP/UDP, enviar mensajes. En este proyecto únicamente se usará la comunicación a través de TCP



Ilustración 3-2 Tarjeta de red Neoway M590

3.1.1.3 Pantalla LCD y expensor de pines

Permite la representación de 32 caracteres divididos en dos líneas de 16 caracteres. Es posible la interfaz directa con la placa Arduino (es decir, conexión en paralelo pin a pin), sin embargo, resulta mucho menos tedioso a la hora del desarrollo y la implementación real dotarlo de una interfaz I2C mediante **un circuito basado en el chip PCF8574**, conocido popularmente como “expansor de pines”. De esta manera, en lugar de necesitar 16 cables para la conexión, únicamente nos harán falta cuatro (reloj, datos, alimentación y tierra).

Para identificar al LCD en el bus I2C, como cualquier otro dispositivo con esta interfaz, será necesario dotarlo de una dirección. Soldando los contactos A0, A1 y A2 (en la fotografía se encuentran sobre el potenciómetro azul). De fábrica, todos los contactos se encuentran a circuito abierto, lo que correspondería a la **dirección 39 (DEC)**. En un proyecto que contenga más de un LCD será necesario soldar algún contacto, no siendo el caso que nos ocupa.

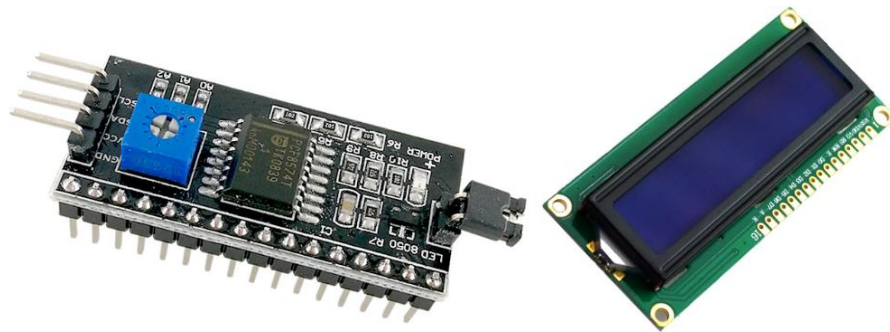


Ilustración 3-3 Izq.: Expansor de pines PCF8574, Der: LCD 16x02

3.1.1.4 Lector de códigos RFID

Se utiliza el modelo RC522. Usado en la autenticación de usuarios en el sistema. A diferencia de los periféricos presentados anteriormente, éste se alimenta a una tensión de 3.3V. Su interfaz con el sistema microcontrolador Arduino es mediante SPI (comunicación serie full dúplex síncrona con bus compartido). Cuando acercamos la tarjeta al sensor, éste, como se verá, comunica al Arduino el código RFID de la misma, formado por 4 bytes (o lo que es lo mismo, 8 dígitos hexadecimales).



Ilustración 3-4 Lector RFID RC522

3.1.1.5 Etapa de potencia

Formada por una fuente de alimentación (se encarga de rectificar el voltaje proveniente de la red eléctrica doméstica) y convertidor DC-DC reductor.

Como ya se explicó en su apartado correspondiente, para la alimentación de los periféricos en sistemas de una complejidad considerable, **es preferible montar un sistema de alimentación aparte, en lugar de conectarlos a la salida de 5V de la placa Arduino**. La fuente de alimentación tiene como salida una tensión aproximadamente continua (siempre existe un rizado estrictamente hablando) de 12 V, mientras que, con el regulador, podemos reducirla hasta los 5V, que es la que buscamos para los periféricos. **Es importante cortocircuitar el nodo de tierra (GND) de la placa Arduino, con la de la F.A.**, de lo contrario podría haber problemas en la comunicación serie.

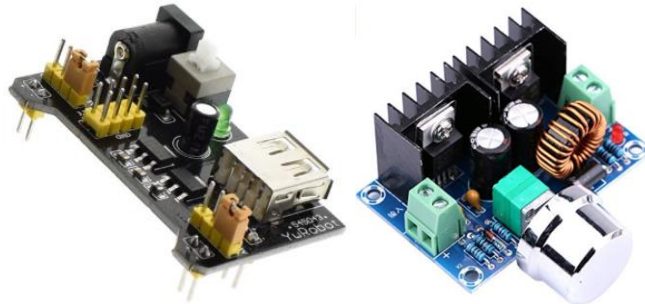


Ilustración 3-5 Fuente de alimentación y convertidor regulable

3.1.1.6 Relé

Es el dispositivo por el cual **se puede habilitar o deshabilitar** (actuando desde la placa Arduino) **el flujo de corriente hacia la carga conectada por el usuario**. Cuenta con tres pines a cada lado. A un lado (los pines machos) se encuentra el pin de alimentación a 5 voltios, el pin de tierra y el de control. Por el otro, los que corresponden al flujo eléctrico que queremos controlar.

Con el pin de control (típicamente denotado en el circuito como IN o IN1, IN2...), podemos actuar sobre el dispositivo. Con un nivel bajo (valor de pin digital LOW en lenguaje de programación Arduino) el relé permitirá el paso de corriente, iluminándose un diodo LED de color rojo en el dispositivo. De manera análoga, si colocamos el pin de control a nivel alto (HIGH) el paso de corriente se interrumpirá hasta nueva orden, apagándose el diodo LED.



Ilustración 3-6 Relé de un solo contacto

3.1.1.7 Cuadro eléctrico

Incluye los elementos necesarios para la seguridad y medición del consumo eléctrico durante el ciclo de recarga, amén del soporte para la conexión de la carga (dispositivo del usuario del servicio).

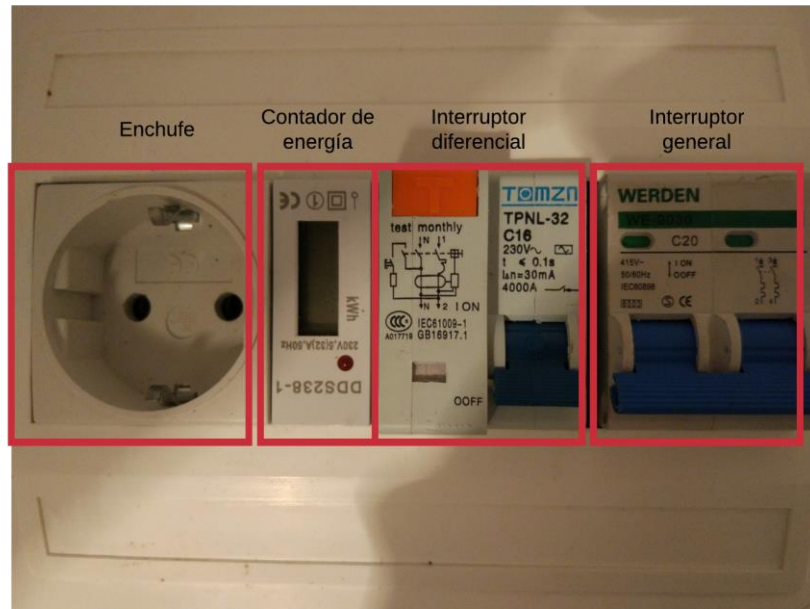


Ilustración 3-7 Cuadro eléctrico completo

Está formado por los siguientes elementos:

- Interruptor general Werden WE 9030: Para cortar de modo manual la corriente al resto del sistema. Está pensado para funcionar con voltajes de hasta 415V en corriente alterna con una intensidad de 20A. Cumple con la norma UNE-EN 60898.
- Interruptor diferencial TPNL-32: Protege la instalación frente a cortocircuitos producidos en la carga, cortando el paso de corriente en estos casos.
- Contador de energía DDS238-1: El elemento más importante desde el punto de vista del sistema. Emite un breve pulso al contabilizar una determinada cantidad de energía. Según las especificaciones, 1600 pulsos equivalen a una energía de 1kWh, por lo que:

$$\frac{1 \text{ kWh}}{1600 \text{ pulsos}} = \frac{1000 \text{ Wh}}{1600 \text{ pulsos}} = \boxed{0.625 \frac{\text{Wh}}{\text{pulso}}}$$

Esta relación será usada en el software para el cálculo de la energía (o cálculo aproximado, mejor dicho). Dicha energía siempre será contabilizada en múltiplos de este número.

- Enchufe tipo hembra para conectar el dispositivo del usuario.

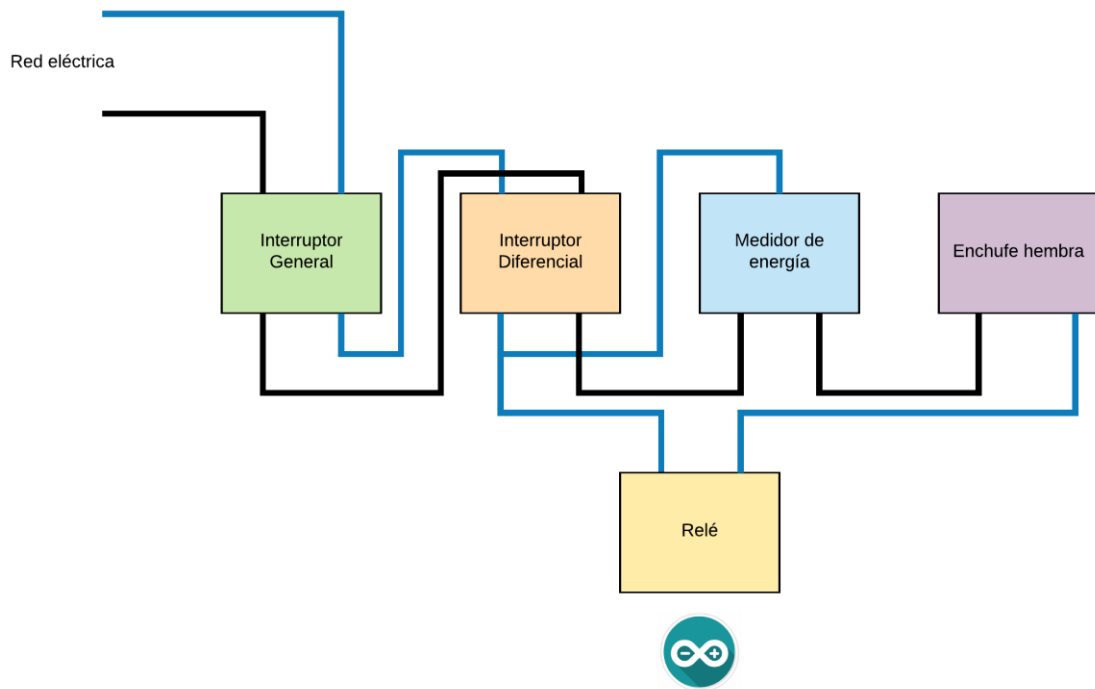


Ilustración 3-8 Esquema de conexión del cuadro eléctrico

El esquema de conexión es muy sencillo, simplemente los dispositivos en cascada uno detrás de otro, excepto el medidor de energía. Su cableado es muy similar a la de un vatímetro, por un lado, necesita tener parte de su circuitería en paralelo, para medir voltaje, y otra parte, en serie, para medir intensidad.

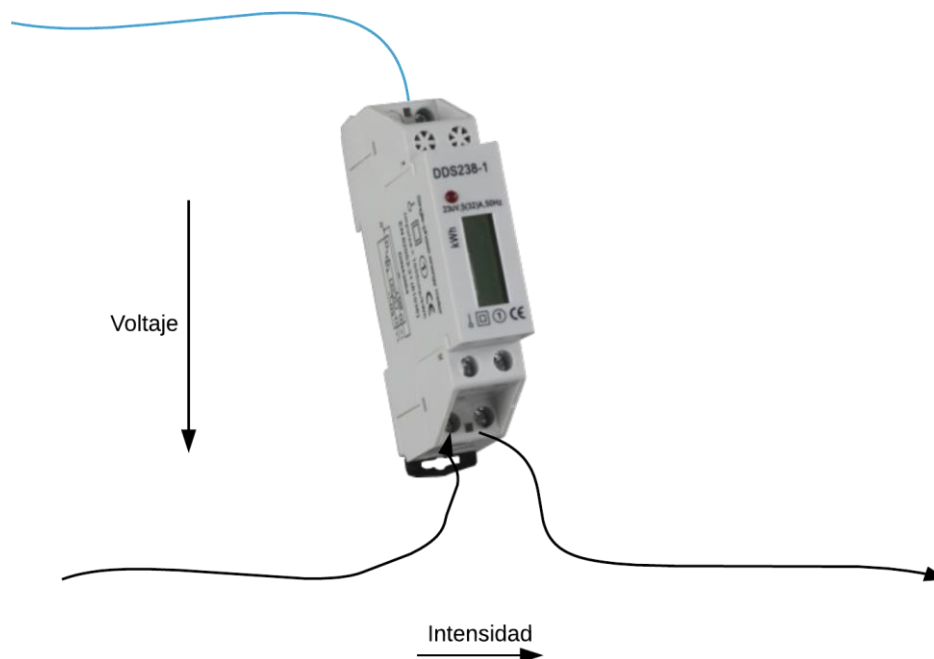


Ilustración 3-9 Medidor de consumo

3.1.1.8 Diagrama de bloques del cliente

A modo de resumen de lo anteriormente explicado, se adjunta un diagrama de bloques en el que se pueden distinguir la interconexión entre la placa Arduino y los diferentes periféricos.

Se ha usado una simbología para representar el sentido de la comunicación. Se pueden apreciar tres tipos de comunicación:

- Comunicación bidireccional: Cuando la placa espera recibir y enviar datos por esa interfaz (p.e. la tarjeta de red GPRS).
- Comunicación con la placa siempre como emisora. Dicho de otra manera, son dispositivos sobre los que la placa actúa. Desde otro punto de vista son ‘salidas al sistema’. Por ejemplo, el relé.
- Comunicación con la placa siempre como receptora. Por ejemplo, el medidor de consumo.

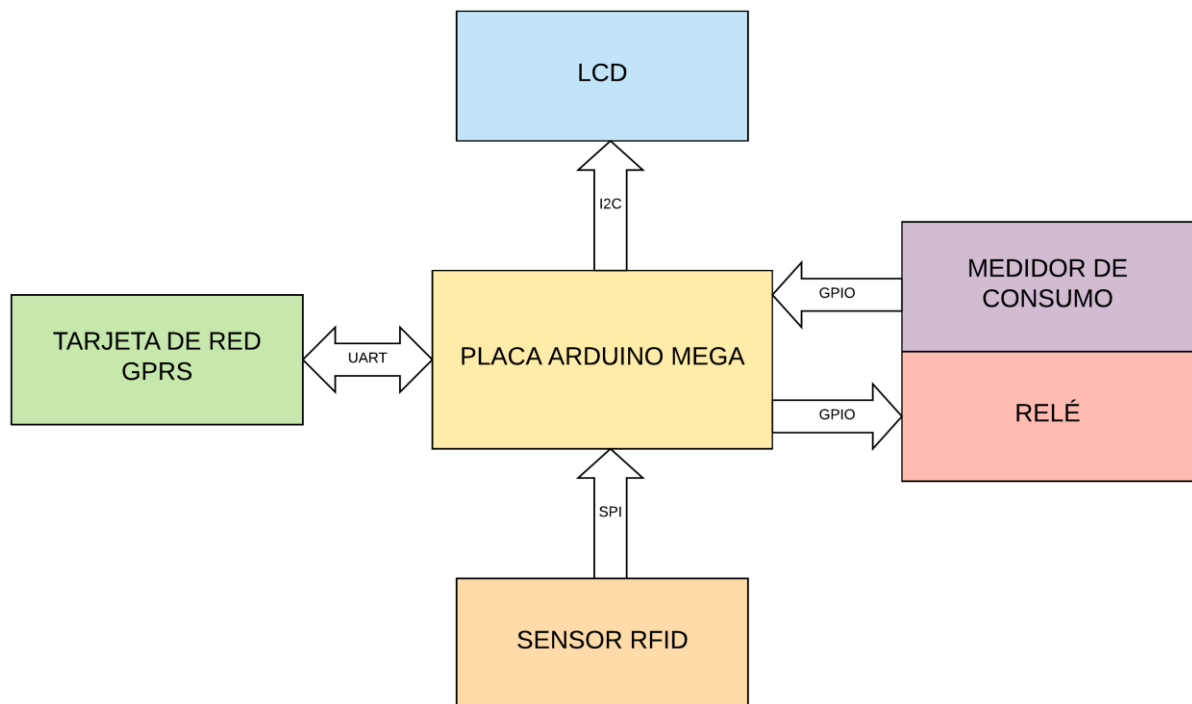


Figura 2 Diagrama de bloques del cliente

En apartados siguientes se profundizará sobre cada una de las conexiones y cómo el microcontrolador gestiona esa interfaz.

3.1.2 Programa en lenguaje Arduino

En esta sección se detallará cómo se ha programado la placa para cumplir la especificación.

3.1.2.1 Subrutina setup

En el código de cualquier sketch Arduino podemos distinguir dos funciones predeterminadas, una de ellas (función `setup`) se ejecutará una única vez al dar comienzo el programa. Se usa, entre otras cosas, para inicializar periféricos y conexiones que se usarán en el programa. En nuestro caso:

- Conexión asíncrona con el monitor serie del software Arduino del ordenador, mediante la función **`serial.begin(X)`**, donde **X** representa la velocidad de esta en baudios. Debe coincidir con la indicada en el monitor serie, de lo contrario los datos recibidos del Arduino no se visualizarán correctamente en el ordenador.
- Conexión asíncrona full dúplex con la tarjeta de red GPRS mediante UART. De igual manera, al no existir señal de reloj, **la tasa de transmisión debe ser conocida de antemano** (de no ser así el receptor no muestrearía todos los bits a tiempo). Según el catálogo del fabricante Neoway, velocidad por defecto son 115.200 baudios (se puede configurar mediante comandos), aunque se recomienda probar antes de implementar el sistema completo, pues puede variar. Se emplea una función similar al monitor serie, en este caso **`serial1.begin(X)`**.
- Conexión serie síncrona SPI con el lector de RFID e inicialización de este. A diferencia de los dos periféricos anteriores, el lector de huellas recibe señal de reloj de la placa Arduino, por lo que no necesitamos especificar la velocidad de la comunicación correspondiente. Las líneas SPI se inicializan con la función **`SPI.begin()`** y con la función **`mfr522.PCD_Init()`** se realizan la operación de inicialización.
- Configuración de los pinos de entrada y salida para propósito general, y si procede, asignarles valor. En nuestro caso hay dos, **el que conecta el medidor de energía consumida** con la placa Arduino (definido como entrada) y el de **actuación sobre el relé**, que, desde el punto de vista de la placa, es salida. La configuración de los GPIO (*General Purpose Input Output*) en Arduino se realiza con la función **`pinMode(npin, modo)`**. En el caso del relé, además, hay que asignarle el valor **HIGH**, mediante la sentencia **`digitalWrite(npin, HIGH)`**, que corresponde, cualitativamente, **a no permitir el paso de corriente**.
- Inicialización y configuración del LCD. Tendremos que tener previamente creada una instancia del objeto `LiquidCrystal_I2C`, llamada `lcd`, amén de conocer la dirección que posee en el bus I2C. Luego se inicializa el LCD mediante la función **`lcd.begin()`**, que toma como parámetros las dimensiones (en número de caracteres) del display. Una vez inicializado, se habilita la iluminación del LCD con la función **`lcd.setBacklight(HIGH)`**.
- Además, se inicializan los punteros a función que marcan la transición entre estados.

3.1.2.2 Procesamiento de mensajes provenientes de la tarjeta de red

Con el fin de **poder identificar los comandos que la tarjeta de red nos envía**, bien como respuesta a uno que nosotros mismos hayamos introducido (o no, como veremos), se ha programado la siguiente estructura dentro de la función `serialEvent1`.

NOTA: La función anterior es llamada automáticamente por el microcontrolador implementado en la placa Arduino cada vez que llega un dato nuevo al buffer de recepción asignado a la comunicación serie número uno (pinos TX1 y RX1 de la placa).

De esta manera no tenemos que estar continuamente comprobando el buffer buscando si hay algún dato nuevo.

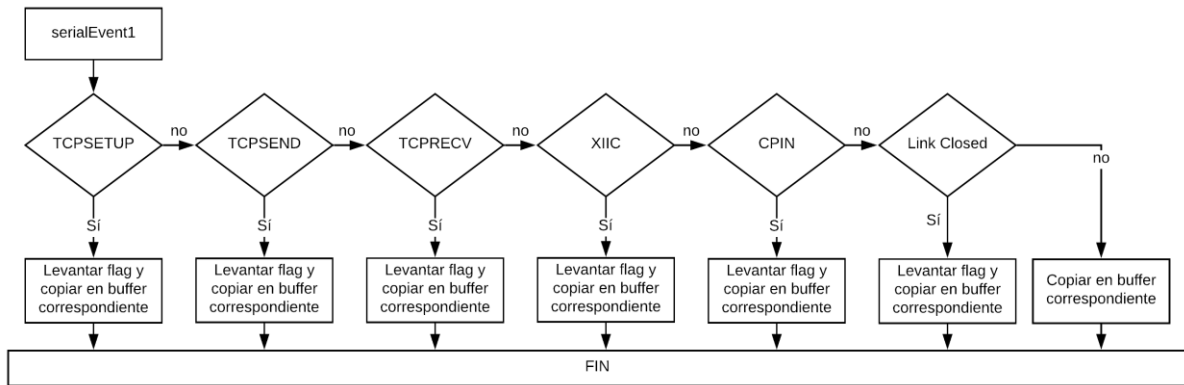


Figura 3 Diagrama de flujo de la función serialEvent()

El uso de flags es necesario porque, adelantándonos un poco a apartados siguientes, cuando el programa espera recibir algún dato del servidor TCP, estará atento a su bandera correspondiente para consultar el buffer ignorando (esa parte del programa) cualquier otro dato entrante.

| Flag que se levanta | Buffer en el que vuelca el contenido para procesamiento por parte del programa | Descripción |
|---------------------|--|--|
| TCPCONN_FLAG | bufferCnxnTCP | Resultado de un intento de establecimiento de conexión TCP |
| TCPTX_FLAG | ACKSalidaTCP | Resultado de un intento de envío de un mensaje al servidor. |
| TCPRX_FLAG | TCPBufferRX | Contiene un mensaje recibido del servidor, junto a su longitud |
| NETSTAT_FLAG | netStatBuffer | Resultado de una comprobación de conectividad a la red GPRS |
| PINCHECK_FLAG | pinCheckBuffer | Resultado de la comprobación del bloqueo de la SIM |
| TCPERROR_FLAG | (no es necesario) | La conexión TCP se ha roto |

Tabla 1 Buffers y flags de gestión del bus serie

De esta manera el flujo principal del programa únicamente tiene que revisar la bandera (flag correspondiente) cuando espere alguna respuesta del módulo. En el texto inferior y la figura se ejemplifica este procedimiento:

1. El Arduino pone en el buffer para transmisión serie el comando que corresponda a la operación que quiere realizar (en este caso para saber si se dispone de conexión a la red “AT+XIIC?”). Tras una corta espera, transmite.
2. Se queda esperando la respuesta, es decir, comprobando periódicamente el flag correspondiente.
3. Cuando la función serialEvent1 procesa el mensaje, levanta la bandera NETSTAT_FLAG para indicarle al proceso principal que ha llegado la contestación a su petición.
4. Se “firma” el acuse de recibo bajando la bandera y analizando el contenido (volcado en el buffer que corresponda).

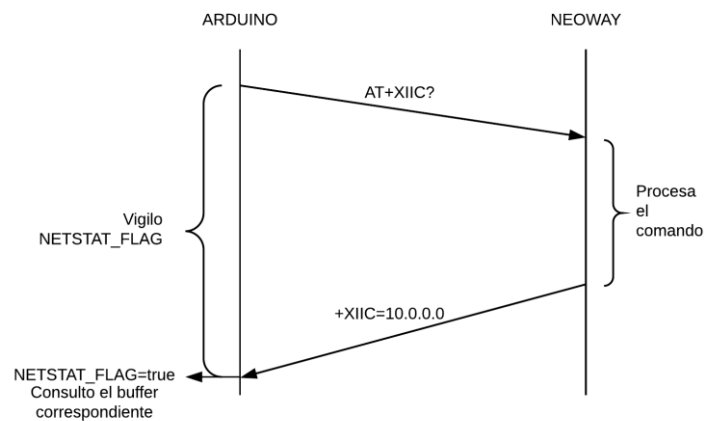


Figura 4 Operación básica de comprobación de conectividad

3.1.2.3 Inicialización

Cuando se ejecuta el programa, el primer paso es inicializar la tarjeta de red:

- Deshabilitar el “eco” en los comandos.
- Si tenemos acceso a la tarjeta SIM, es decir, la tarjeta SIM no se encuentra bloqueada por un código PIN ni PUK.
- En caso afirmativo, configurar parámetros y comprobar si se dispone de conexión a la red GPRS.
- Cuando todo lo anterior haya tenido éxito, establecer conexión TCP al servidor.

Por defecto la tarjeta de red está configurada para que, cuando reciba un comando, **repetirlo precediendo a la respuesta que envíe**. Con el objetivo de evitar la recepción de información altamente redundante, es preferible deshabilitarlo, para que únicamente mande la respuesta. Esto se consigue con el comando “ATE0” (abreviatura de “AT Command Echo”). En contrapartida, el comando “ATE1” vuelve a configurar el eco del comando recibido.

La comprobación del desbloqueo de la tarjeta SIM se realiza a través del comando “AT+CPIN?”. Las respuestas posibles a este comando son tres. Si la respuesta es “+CPIN: READY”, **el módulo se encuentra listo para ser usado**, las otras dos posibles respuestas son “+CPIN: SIM PIN”, la tarjeta se encuentra bloqueada por un código PIN, y si la respuesta es “+CPIN: SIM PUK”, por el código PUK (resultado de haber fallado al introducir el código PIN tres veces).

En este proyecto se ha dado por hecho que la tarjeta SIM con la que se trabaja está desbloqueada, si no fuera así el caso sería tan sencillo como proporcionarle al módulo “**AT+CPIN=“XXXX”**” donde XXXX es el número de cuatro dígitos correspondiente al PIN.

Una vez que nos hemos cerciorado del desbloqueo de la SIM **se puede proceder a la configuración de los parámetros de la red GPRS**, como son el nombre del APN (*Access Point Name*) al que nos vamos a conectar, la torre de protocolos a usar y el establecimiento de la conexión a nivel de capa de enlace (protocolo PPP, *Point to Point Protocol*). No confundir con conectividad a nivel 3 (IP) o conectividad a nivel 4 (TCP). El procedimiento de conexión sigue un proceso ascendente en la torre OSI.

Para el nombre del APN, definido en el manual del fabricante como “configuración del formato a usar por el protocolo PDP” **haremos uso del comando “AT+CGDCONT=<cid>, <tipo>, <apn>”**. En el campo “cid” podemos poner un número a libre elección, siempre que sea mayor o igual que uno, en el campo “tipo”, el tipo de paquete PDP a usar, en nuestro caso, IP. Por último, en APN, tendremos que poner el que nos corresponda según el operador al que pertenezca la tarjeta SIM a usar. Algunos ejemplos son (en caso de duda se puede consultar al operador):

| Operador | APN |
|----------|---------------|
| Movistar | telefonica.es |
| Vodafone | airtelwap.es |
| Orange | orangeworld |
| Jazztel | jazzinternet |

Tabla Ejemplos de operadores de red móvil y sus respectivos APN

NOTA: El APN (Nombre de punto de acceso en castellano) es un nombre, que, al resolverlo mediante el protocolo DNS proporciona **la dirección de una pasarela que hace de puente entre la red móvil y (típicamente, aunque no siempre) la interred pública**. Es, por tanto, necesario conocer el nombre o dirección IP de esta pasarela para poder establecer cualquier comunicación con algún nodo exterior a la red del operador.

Suponiendo que nuestra tarjeta SIM es de la operadora Movistar, el comando completo sería: **“AT+CGDCONT=1, "IP", "telefonica.es”**, prestando especial atención a entrecomillar bien tanto “IP” como el nombre del APN. A lo que la tarjeta de red simplemente responde “OK” si no se ha corrompido el mensaje, aunque el nombre del APN especificado no exista o sea erróneo.

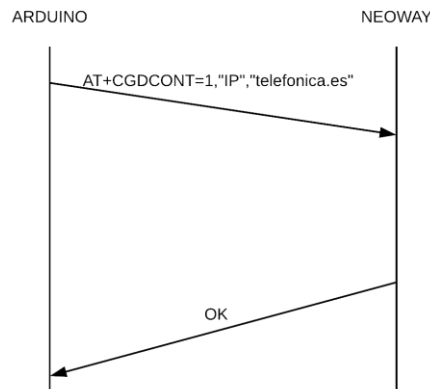


Figura 5 Configuración del APN

El siguiente paso es elegir la arquitectura de protocolos a usar. Sin lugar a duda, por compatibilidad con el resto de la red y el servidor, **escogemos la torre de protocolos TCP/IP**, quedando plasmado en el comando **“AT+XIISP=1”**. De igual manera al comando anterior, la tarjeta de red devuelve siempre “OK”.

Tras configurar el nombre del APN y la torre de protocolos, debemos entonces intentar el establecimiento de conexión a nivel de capa de enlace, que en GPRS está ocupada por el protocolo PPP (*Point to Point Protocol*). Esta tarea se lleva a cabo a través del comando **“AT+XIIC=1”**

NOTA: Los tres comandos anteriores (**AT+CGDCONT=1, "IP", "telefonica.es", AT+XIISP=1 y AT+XIIC=1**), al ser respondidos únicamente con un “OK”, no requieren de procesamiento de respuesta.

Una vez definidos los parámetros de la conexión, **toca comprobar si se ha establecido correctamente**, es decir, si la red GPRS del operador nos ha asignado una IP “válida” (entiéndase válida como diferente de “0.0.0.0”). Es importante notar que esta IP asignada es “privada”, lo que significa que no tiene validez fuera de la red del operador (de igual manera que las direcciones IP privada de las redes locales sólo son válidas en ese ámbito).

El comando usado para comprobar la conectividad es muy similar al último, se trata de **AT+XIIC?** (nótese que el signo de interrogación cuando se solicita información a la tarjeta de red, de igual manera que el comando de

comprobación de desbloqueo de la SIM.

La respuesta del módem a este comando se captura, como se vio en el apartado correspondiente, en la función **serialEvent1**, que al detectar la cadena de texto “+XIIC” en los datos que llegan por la línea serie **levanta la bandera correspondiente** para indicarle al programa principal que la petición ha sido respondida. Al recibir la respuesta hay dos posibles casos:

- La conexión se ha establecido correctamente, en este caso, la respuesta será del tipo **+XIIC: 1, A.B.C.D**, siendo la IP diferente de 0.0.0.0. Se puede proseguir con la ejecución normal del programa.
- El intento de establecimiento de conexión ha fallado, siendo **+XIIC: 0, 0.0.0.0** la respuesta. Hay que reintentar el procedimiento de conexión.

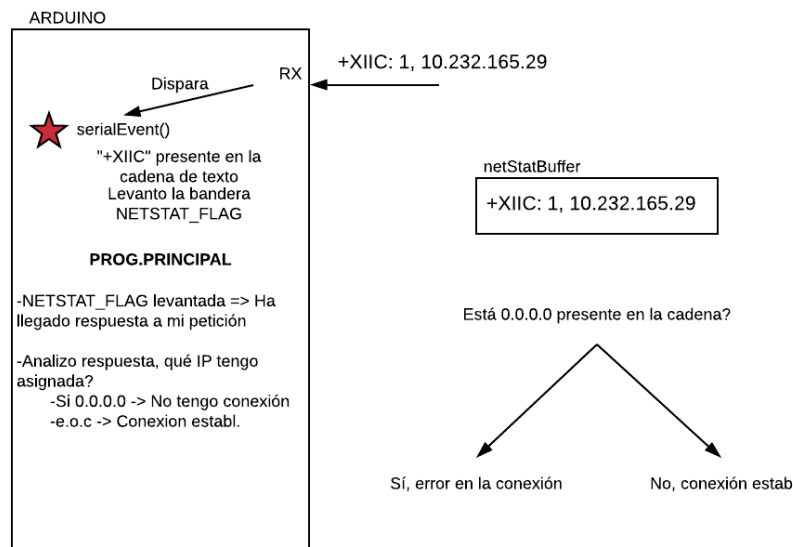


Figura 6 Comprobación de la conexión a la red GPRS

Una vez recibida la respuesta, y teniéndola almacenada en la cadena `netStatBuffer`, hemos de, como indica la figura, comprobar si contiene 0.0.0.0. Esto se puede realizar fácilmente con el método (no confundir con función) `indexOf`. Este método comprueba si la cadena pasada por parámetros pertenece a la cadena en la cual se invoca el método. De ser parte de ella, **devuelve el índice de comienzo** (recordar que en Arduino el primer elemento de una matriz es el número cero), de lo contrario, **devuelve un valor inferior a cero**.

A la hora de establecer conexión TCP tendremos que andar con ojo sobre qué tupla (IP: Puerto) configuramos en el comando correspondiente. Si el equipo ejecutando el proceso se encuentra en una red local (LAN) es muy probable que router doméstico esté haciendo NAT (*Network Address Translation*).

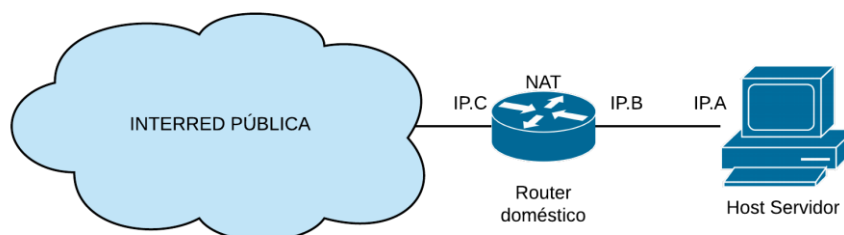


Figura 7 Arquitectura de red en el lado del servidor

La figura superior ilustra lo que queremos describir. **Una persona sin los conocimientos necesarios en redes de ordenadores simplemente pondría como IP del servidor en el comando IP.A.** Como es obvio, dicha petición de conexión nunca tendría éxito. **Habiendo realizado la configuración en el router que se detalla en el anexo correspondiente**, la dirección a incluir sería la de la interfaz pública del router (**IP.C** en la figura). El número de puerto a incluir también se detalla en el anexo.

Por tanto, el comando sería **AT+TCPSETUP=0, IP.C, port**. El programa, tras enviar el comando a la tarjeta de red, deberá esperar a la respuesta de este, indicando si la conexión al servidor ha tenido éxito o no.

- Si se ha logrado establecer la conexión, la tarjeta responderá **+TCPSETUP:0, OK**
- En otro caso (por ejemplo, si no hay ningún socket escuchando en ese puerto o la dirección IP facilitada es inalcanzable) **+TCPSETUP:0, FAIL**

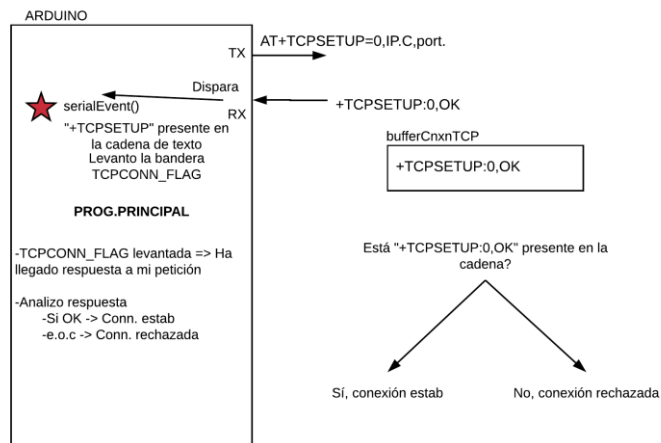


Figura 8 Establecimiento de conexión con el servidor

Con la conexión TCP establecida se da por terminado el proceso de inicialización del cliente. Debajo se resume dicho proceso en un diagrama de flujo

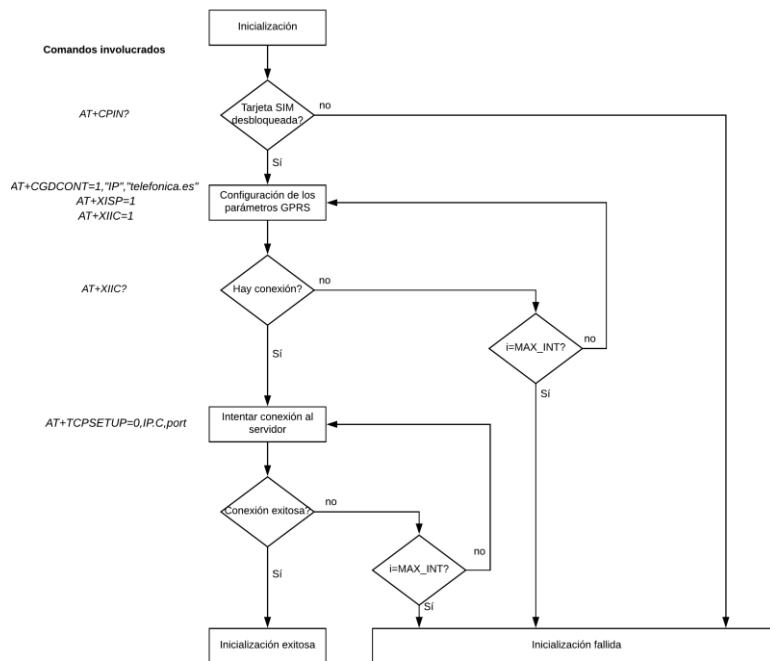


Figura 9 Tabla resumen del proceso de inicialización

3.1.2.4 Ciclo de recarga

3.1.2.4.1 Espera de asentimiento del servidor y de código RFID

Cuando el servidor está listo para recibir el código RFID del usuario envía el mensaje “WAITING FOR RFID CODE”. La recepción de este mensaje, en el lado cliente de la aplicación distribuida, a nivel funcional, **habilita la subrutina de lectura RFID_FLAG** mediante el levantamiento del flag correspondiente (se describirá más adelante esta subrutina).

A modo de recordatorio, como cualquier mensaje proveniente del buffer de recepción TCP, al ser recibido por la placa Arduino proveniente de la tarjeta de red, levanta el flag `TCPRX_FLAG`. Al detectar esta bandera en nivel alto, el programa sabe que el contenido está en la cadena de texto `TCPBufferRX`, por lo que, con la ya mencionada función `indexOf` puede averiguarse si la cadena contiene el mensaje que estábamos esperando.

Cuando la subrutina encargada de la lectura de la tarjeta RFID ha terminado, es decir, ha completado la lectura de manera exitosa levanta la bandera de fin de lectura (`RFID_completed_read`) para informar al flujo principal del programa, estando el código leído en la cadena de texto `RFID_salida`.

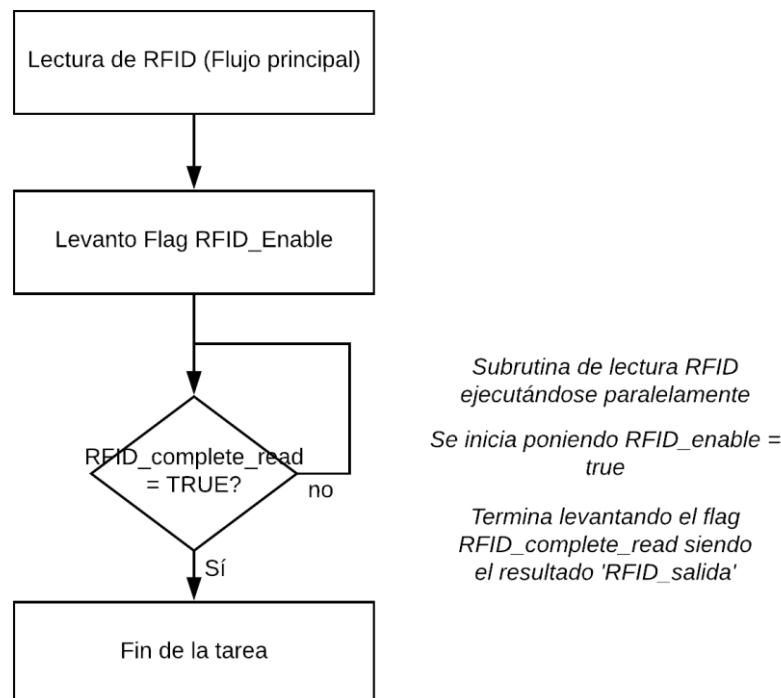


Figura 10 Diagrama de flujo resumen del proceso de espera de código RFID

3.1.2.4.2 Envío de código RFID al servidor y respuesta

Tras haber leído y procesado el código RFID, la siguiente tarea a realizar es el envío al servidor para su cotejo en la BBDD.

En la tarjeta de red, el envío de un mensaje a través de una conexión TCP se realiza en dos pasos.

- En el primero se indica el **número de la conexión** (cero o uno) que se especificó en la orden de establecimiento, y el **tamaño, en bytes, del mensaje a entregar**, o, lo que es lo mismo, el número de caracteres ASCII. Recordemos que cada carácter, en ASCII puede ser representado mediante 8 bits (1 byte).
- Cuando la tarjeta de red responde con el carácter '>' nos está indicando que está lista para recibir el flujo de bytes (caracteres) a enviar al otro extremo de la conexión TCP. **Es importante destacar que, aunque hayamos introducido el número indicado en el paso anterior, no se enviará nada hasta que no introduzcamos el carácter de retorno** (en código ASCII equivale a '`0x0D`').

Tras el carácter de retorno, con un breve período de procesamiento interno de la tarjeta de red, ésta responderá por UART, indicándonos el resultado del intento de envío, existiendo dos posibilidades

- Éxito en la transmisión. La respuesta será del tipo **+TCPSEND: 0, N**, siendo N el número de bytes enviados. TCP ya incorpora, de por sí, algoritmo para cerciorarnos que el segmento haya sido entregado de manera íntegra al receptor.
- Fallo en la transmisión. Una posible causa es que se haya perdido la cobertura de la red GPRS. En este caso el comando de respuesta será **+TCPSEND: Error**.

Luego del envío del código RFID procesado al servidor, éste hará el cotejo con la tabla correspondiente de la BBDD (ya se verá en el apartado correspondiente cómo) **emitiendo una respuesta**, según haya habido coincidencia en la consulta, o no.

- Si el servidor ha encontrado que el código RFID enviado **consta como asociado a un usuario** registrado en la BBDD, enviará, a través de la conexión TCP, el mensaje **'QUERY SUCCESS'**.
- De lo contrario, el código no corresponde con ningún usuario registrado, el mensaje de respuesta será **'QUERY FAILED'**. Llegados a este punto, **no hay más acciones que realizar en la sesión actual** (no se permite la recarga de usuarios no autenticados), simplemente volveríamos al punto donde se solicita un nuevo código (el servidor enviaría **'WAITING FOR RFID CODE'**).

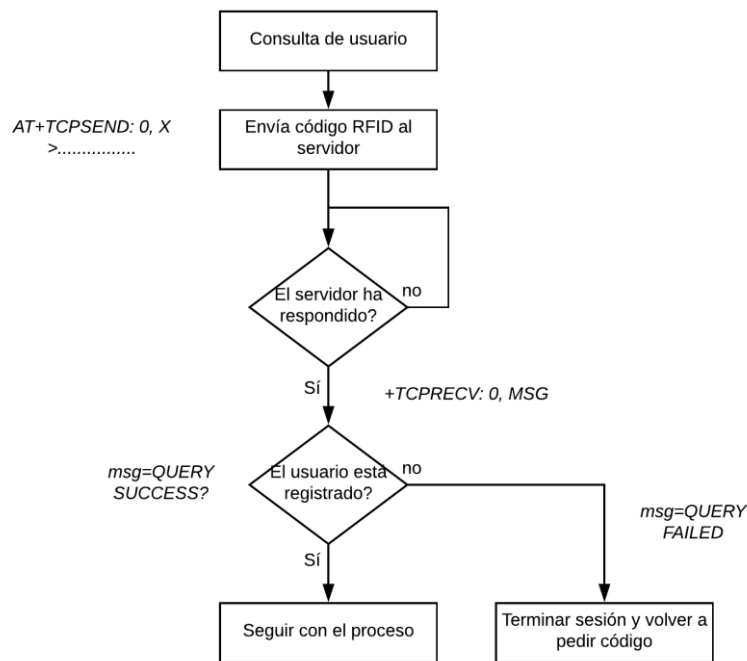


Figura 11 Diagrama de flujo del envío y respuesta del código RFID al servidor

3.1.2.4.3 Recarga e informe al servidor

Mediante una subrutina, que se detallará en su preciso momento más adelante, se realiza un seguimiento del tiempo que el usuario está haciendo uso del suministro eléctrico de la instalación. El proceso, desde el punto de vista del flujo principal del programa es el siguiente:

1. Se habilita el paso de corriente eléctrica hacia el enchufe, mediante el establecimiento del **pin de control del relé a nivel bajo**.
2. Se habilita la interrupción hardware por medio de la función correspondiente en Arduino (**attachInterrupt**).
3. Cuando se computa un pulso, la subrutina coloca a nivel alto la bandera correspondiente (**NEWPULSE_FLAG**) para notificar al flujo principal, que al detectarla en nivel alto envía un mensaje al servidor (de contenido únicamente **'PULSO'**). De este modo, para mayor seguridad, la cuenta se lleva en el servidor, en lugar de

en el cliente (podría simplemente enviarse al final de la recarga un mensaje ‘X pulsos’)

4. Cuando la subrutina detecta que ha transcurrido **un intervalo de tiempo** (configurable desde código) **desde el último pulso detectado**, o desde el inicio del proceso, da por concluido el ciclo, esto es, corta el paso de corriente hacia el enchufe, forzando un nivel alto en el pin de control del relé, enviando un ‘FIN’ al servidor. **Esto se detecta desde el flujo principal del programa mediante el nivel alto de la bandera FINRECH_FLAG**, puesta desde la subrutina.

Habiendo completado el ciclo, no quedarían más operaciones que hacer en el lado del cliente, por lo que quedaríamos a la espera de que el servidor nos enviara un nuevo “WAITING FOR RFID CODE” para iniciar un nuevo ciclo.

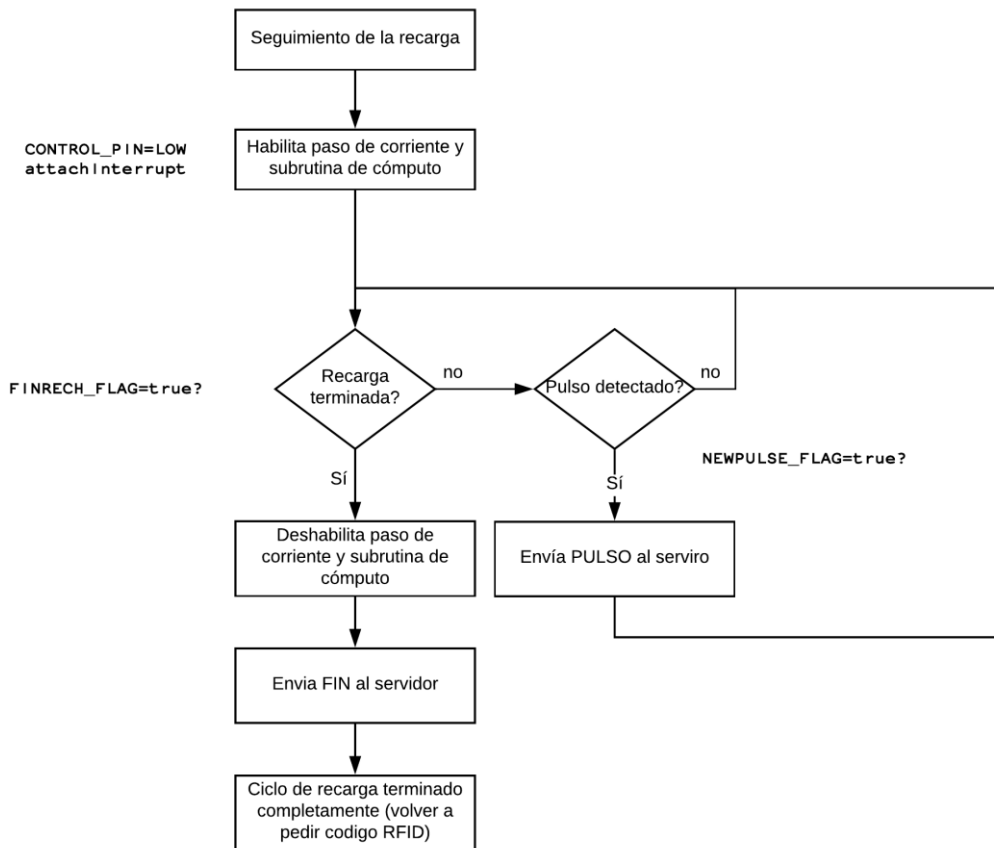


Figura 12 Seguimiento de la recarga desde el flujo principal de programa

3.1.2.5 Subrutina: Lectura del código RFID

En el momento de describir el flujo principal del programa no profundizamos lo suficiente para detallar el proceso de la lectura del código RFID. Resumidamente, lo que dijimos fue:

- Se inicia cuando el programa principal ha colocado en true la bandera **RFID_flag**. No está de más decir que la subrutina la baja a modo de asentimiento, **aunque es más correcto decir que es para no provocar un bucle infinito**.
- Cuando ha leído de manera correcta una tarjeta pone a nivel alto la bandera **RFID_completed_read**, quedando el código leído en formato string almacenado en la variable **RFID_salida**.

Esta subrutina está fuertemente basada en la librería MRFC552. Algunas funciones y variables importantes propias de la librería son:

| Función/Variable | Descripción |
|------------------------------|---|
| PICC_IsNewCardPresent | Función que sirve para comprobar si el sensor RFID detecta algún chip cerca, devolviendo TRUE si se da el caso, FALSE si no encuentra ningún chip en su rango. |
| PICC_ReadCardSerial | Una vez que el sensor ha detectado una tarjeta cerca, esta función lee su valor devolviendo una tabla de dígitos hexadecimales almacenados en la variable uid. uidByte |
| uid. uidByte | Tabla de números en formato hexadecimal, resultado de la lectura de la presente tarjeta. |

Tabla 2 Funciones y variables de la librería de RFID

Para que la lectura de la tarjeta sea posible, deben cumplirse **dos condiciones**:

- En primer lugar, el proceso principal del programa DEBE requerirlo. Esto implica, en tema de código, que haya colocado **RFID_flag** en true.
- Que haya una tarjeta cerca del sensor, como es lógico. Como ya hemos visto, esto se puede averiguar porque **PICC_IsNewCardPresent** devuelve true en ese caso.

Cuando ambas condiciones se han cumplido, podemos llamar a la función para obtener los dígitos hexadecimales **PICC_ReadCardSerial**.

Con las cifras hexadecimales dispuestas en una tabla, queremos, primero, convertir cada una por separado en cadenas de texto, para después concatenarlas.

Arduino cuenta con una función que simplifica mucho la tarea de convertir variables enteras (**integer**) en caracteres (**char**). Esta función se llama **itoa** (*integer to array*). Toma tres argumentos:

- En primer lugar, el dato numérico.
- En segundo lugar, la variable donde ‘depositar’ el resultado.
- Por último, la base en la que se quiere expresar el resultado, en este caso, dieciséis, ya que es un dato hexadecimal.

El siguiente paso es concatenar cada cadena de texto generada anteriormente para formar la cadena unificada que se quiere pasar al flujo principal (por ejemplo, si el resultado de la conversión es ‘aa’, ‘bb’, ‘cc’ y ‘dd’, queremos unificarlo en ‘aa-bb-cc-dd’, para lo cual el método **concat** del tipo de variable string es muy útil.

Para las tareas iterativas, como puede ser ir recorriendo índices de la tabla de bytes para convertirlos a string, se puede usar la propiedad **uid. size**, ya que la función **length** puede presentar fallos con este tipo de datos.

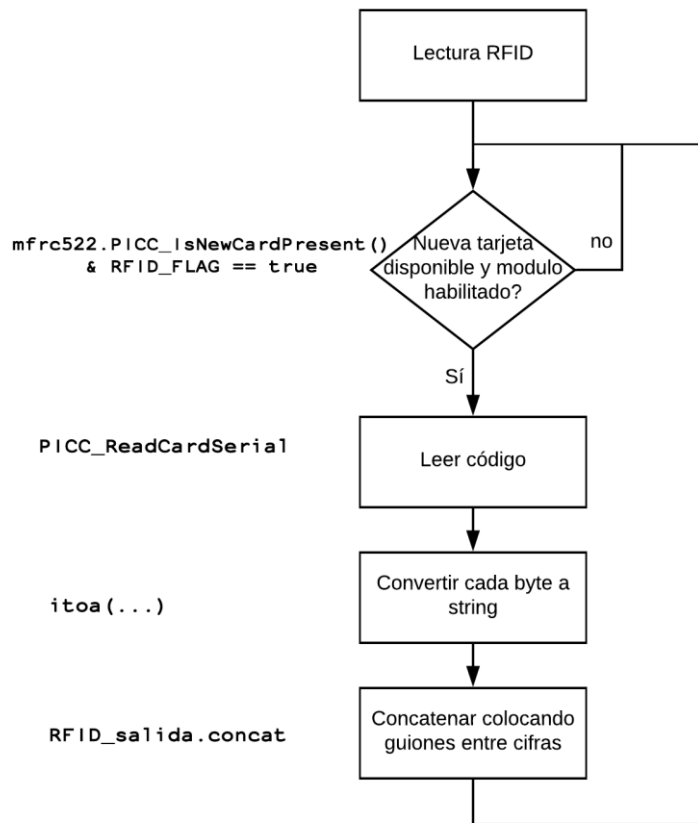


Figura 13 Resumen de la subrutina de lectura de RFID

3.1.2.6 Subrutina: gestión del ciclo de recarga

Por último, se presenta la subrutina que se encarga, de manera muy simplificada, del recuento de pulsos y de informar al flujo principal del programa cuando haya transcurrido una cierta cantidad de tiempo sin nuevos pulsos detectados. Para ello hay que conocer más a fondo la función `attachInterrupt` de Arduino. Esta función sirve para asignar otra función que se ejecute cada vez que ocurre cierto evento en una entrada de la placa. En la literatura de sistemas electrónicos digitales es frecuente referirse a estos eventos como **interrupciones hardware**. Este nombre es debido a que tradicionalmente se ha usado para notificar al microcontrolador que cierto periférico tiene algo que comunicarle.

| | |
|--|---|
| Prototipo: <code>attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)</code> | |
| pin | El número de pin al cual queremos vincular la interrupción, no todos los pines soportan esta funcionalidad. |
| ISR | Función que será llamada cuando se dé el evento descrito a continuación |
| mode | Evento el cual dispara la interrupción. Existen dos tipos de evento, de estado (HIGH/LOW) o de cambio (RISING/FALLING/CHANGE) |

Tabla. Prototipo y parámetros de la función `attachInterrupt`

En terminología no tan estricta, estamos configurando la placa para que cuando pase cierto evento, en cierto pin, ejecute una función que normalmente habrá sido escrita por nosotros. En esta aplicación particular nos interesan los eventos de cambio, siendo indiferente el uso de FALLING o RISING, como se explicará.

Es fácil intuir la utilidad de esta herramienta en el problema. Tenemos un dispositivo (contador energético), el cual emite pulsos, y queremos detectarlos y realizar el procesamiento pertinente. Una solución, aunque lejos de

ser óptima sería una consulta continua (*polling*) al valor del pin conectado al contador, lo cual cargaría a la CPU de tedioso trabajo que podría ser realizado de manera mucho más eficiente.

Sin embargo, no todos los pulsos detectados por la interrupción hardware indican que se está consumiendo energía. El medidor de energía emite pulsos con un período de 20ms indefinidamente hasta que computa la energía necesaria (0.625Wh), entonces emite un pulso de una duración mucho mayor (de más de 100ms). Éste último es el pulso que tenemos que detectar, desechando los demás.

Surge, pues, la necesidad de **temporizar**. Por un lado, para llevar cuenta del **tiempo que llevamos sin tener un pulso nuevo**, y, además, para **estimar el período de un pulso**, es aquí cuando entra en juego la función `millis`.

El planteamiento es bien sencillo:

- Al detectar un pulso nuevo, **tenemos que comprobar su período**, es decir, cuánto hace que detecté el último pulso (válido o no). Si es lo suficientemente grande, será un pulso válido para ser contabilizado. Sin olvidar que hay que reiniciar el contador de tiempo sin pulsos válidos (a efectos de finalización de recarga).
- De no ser un pulso válido por tener un período demasiado corto (típicamente 20ms) tenemos que sumar el tiempo computado al acumulado sin pulsos válidos y comprobar esta cantidad con el tiempo configurado sin pulsos válidos, **de haberla superado, concluimos que el ciclo ha terminado**.

En Arduino las interrupciones hardware pueden ser fácilmente deshabilitadas con la función `detachInterrupt`, para cuando el ciclo haya concluido, no se siga disparando la interrupción.

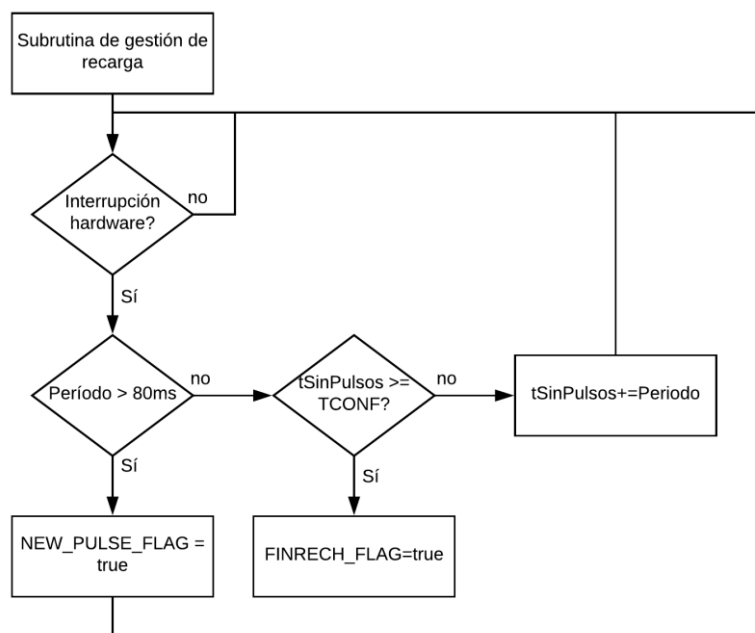


Figura 14 Diagrama de flujo de la subrutina de gestión de recarga

3.2 Programa en lenguaje Python (Lado servidor)

Al otro lado de la comunicación, ocupando el rol que en arquitectura de aplicaciones distribuidas se le llama servidor, se encuentra, en un equipo, se encuentra corriendo un programa escrito en lenguaje Python. Los motivos por lo que nos decantamos por este lenguaje de programación son los siguientes:

- Es un lenguaje interpretado, es decir, no necesita compilación previa a la ejecución. Se podría decir, grosso modo, que “se compila en tiempo de ejecución”.
- Cuenta con librería de funciones para conexión con servidores de correo que usen el protocolo SMTP (*Simple Mail Transfer Protocol*) evitando, así, el tedioso proceso de hacerlo manualmente con sockets TCP.
- Cuenta, también, con librería para transacciones con bases de datos basadas en arquitectura SQL (*Structured Query Language*).
- Es un lenguaje muy intuitivo en materia de sockets, sin ser tan tedioso como el lenguaje C en su manejo.

3.2.1 Ejecución y parámetros en línea de comandos.

Para poner en marcha el servidor, necesitaremos, además, especificar un número en línea de comandos, siendo éste, el número de puerto en el que el socket TCP escuchará (operación de *listening* en inglés) peticiones de conexión. Si no se especifica este valor, el programa detendrá su ejecución, informando del error:

```
C:\Python27>python TFGServer.py
****ERROR****
Uso del script:
>python TFGServer.py + (Numero de puerto TCP de escucha)
C:\Python27>
```

Ilustración 3-10 Ejecución del script sin especificar número de puerto

El número por introducir deberá estar comprendido en el intervalo [0,65535], si, por ejemplo, escribimos un número demasiado alto, dará error al vincular (función *bind*) el socket creado al número de puerto, por lo que no es necesario tratar este error en el código del programa.

```
C:\Python27>python TFGServer.py 70000
Traceback (most recent call last):
  File "TFGServer.py", line 37, in <module>
    s.bind((TCP_IP, TCP_PORT))
  File "C:\Python27\lib\socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
OverflowError: getsockaddrarg: port must be 0-65535.
C:\Python27>
```

Ilustración 3-11 Número de puerto fuera de rango

Este número de puerto deberá ser coherente con la configuración NAT del router, como se explica en el anexo correspondiente, tendrá que coincidir con el puerto LAN (*Local Area Network*) de la tabla de reenvío del router.

3.2.2 Librerías utilizadas en el script

Lo primero que tiene que incluir todo fichero de programación es una lista ordenada de todas las librerías usadas en él, en nuestro caso:

```
import sys
import socket
import mysql.connector
import datetime
import email
import smtplib
```

Importaciones de librerías externas

- **Librería de funciones y parámetros específicos del sistema** (`sys`). Se utilizan dos funciones, `sys.argv` para acceder a los parámetros en línea de comandos pasados al script, y `sys.exit`, para terminar con la ejecución en caso de error irreparable.
- **Librería para las operaciones de red de bajo nivel** (`socket`). Proporciona acceso a la interfaz de sockets (elementos del sistema operativo usados por las aplicaciones del sistema para enviar y recibir información). Algunas de las funciones usadas son la de creación de un socket, vinculación a un puerto, escucha...
- **Librería para la realización de transacciones con bases de datos basadas en MySQL** (`mysql.connector`). Proporciona las herramientas de software necesarias para interactuar con bases de datos (BBDD) basadas en esta tecnología.
- **Librería para el manejo de fechas y horas** (`datetime`). Necesaria para obtener la fecha y la hora actuales en tiempo de ejecución. Se emplea cuando se quiere introducir en la BBDD el momento en el que se ha realizado la recarga.
- **Librería para el manejo y manipulación de emails** (`email`). Esta librería, al contrario de lo que pueda parecer, no se encarga de lo que es su envío (para ello el programador tendrá que importar la librería correspondiente al protocolo que use el servidor de correo), sino de la creación del objeto 'email', con los atributos correspondientes a 'destinatario', 'asunto', 'remite'...
- **Librería específica del protocolo de correo usado** (`smtplib`). Implementa el protocolo de correo SMTP (*Simple Mail Transfer Protocol*), para la comunicación con el servidor de correo (autenticación y envío de correos principalmente). Python incorpora otras librerías similares como `poplib` (protocolo POP, *Post Office Protocol*) o `imaplib` (protocolo IMAP, *Internet Message Access Protocol*).

3.2.3 Inicialización

3.2.3.1 Comprobación del número de parámetros

Lo primero que tenemos que hacer es comprobar el número de argumentos pasados por línea de comandos al script. Éste deberá ser exactamente igual a dos (el primero es el nombre del programa):

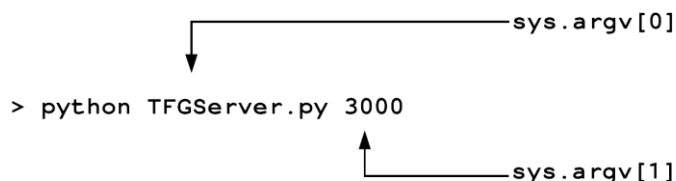


Figura 15 Acceso a los parámetros de ejecución del script

`sys.argv` es una tabla que contiene los parámetros de ejecución, para saber si el número de parámetros es igual a un número tenemos que usar la función `len`. Si el resultado no es igual a dos, el programa fuerza la salida, como hemos visto en un apartado anterior, mostrando el siguiente mensaje:

```
C:\Python27>python TFGServer.py 3000 a
****ERROR****
Uso del script:
>python TFGServer.py + (Numero de puerto TCP de escucha)
```

Error que se muestra cuando el número de parámetros no es correcto

3.2.3.2 Parámetros del socket

Una vez comprobado el número de parámetros y verificado que es correcto es hora de darle los valores de configuración que luego le asignaremos al socket. Ellos son el par dirección IP y puerto. La dirección IP puede dejarse en blanco (y, de hecho, se recomienda).

Llegados a este punto debemos considerar la posibilidad de que el parámetro no se corresponda con un número, o, al menos, no en su totalidad (por ejemplo, 123a4), **en ese caso la función `int`** que convierte variables de cadena de caracteres a enteros, **arrojaría error** (en concreto, `ValueError`), que sería capturado en una estructura `try/except`.

```
C:\Python27>python TFGServer.py 12a45
Error en el número de puerto, debe estar formado únicamente por cifras
```

Con el número de parámetros y el tipo comprobado, toca crear el socket. En Python, existen dos tipos de sockets, definidos en el momento de su creación, los de tipo `SOCK_STREAM` y los de tipo `SOCK_DGRAM`, asociados a los protocolos TCP y UDP respectivamente. Por tanto, tendremos que especificar en la función que queremos un socket tipo `SOCK_STREAM` ya que nuestra comunicación es a través de TCP. **Esta tarea de creación de socket se realiza con la función y librería homónima**

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

Tras haberlo creado toca vincularlo a una dirección IP y puerto. Como se dejó caer párrafos atrás, se puede dejar el campo de dirección en blanco (y de ser así, se tomará del sistema operativo). Esta tarea se lleva a cabo con la función `bind`.

3.2.3.3 Conexión a la base de datos

Empleando la librería `mysql.connector.connect` podemos establecer conexión a la base de datos. Los parámetros de la conexión se especifican como argumentos, como también la base de datos específica a la que se va a acceder:

```
cnx = mysql.connector.connect(user='root',
password='root',host='127.0.0.1', database='usersdb')
```

El caso anterior correspondería al caso de tener la BBDD alojada en el mismo equipo (la dirección especial `127.0.0.1` identifica a uno mismo). También es necesario especificar tanto el identificador de usuario como la clave, permitiendo la convivencia de varios usuarios, cada uno con diferentes permisos de lectura y escritura.

Pueden darse tres tipos diferentes de errores:

- No hay ninguna BBDD en la dirección especificada: En este caso la sentencia de establecimiento de conexión arrojará una excepción del tipo `UnicodeDecodeError`, apareciendo en consola el siguiente error (cambiando la dirección IP a, por ejemplo `172.0.3.1`):

```
C:\Python27>python tfgserver.py 3000
Error conectando a BBDD, revise la dirección IP
```

- No existe la combinación de usuario/clave aportada o bien no existe una BBDD con el nombre aportado en el host especificado. Por ejemplo, cambiando el nombre de la base de datos a 'bddprueba':

```
C:\Python27>python tfgserver.py 3000
```

```
Error conectando a BBDD, la BBDD no existe o la combinación usuario/pass
es incorrecta
```

3.2.3.4 Espera de conexión entrante

Una vez que se han inicializado correctamente tanto la conexión a la BBDD toca poner el socket TCP en modo 'escucha' para atender la petición de conexión de la estación remota. Esto se realiza con la función `listen` y `accept`.

La función `accept` vincula la conexión aceptada, valga la redundancia, con otro socket distinto al de escucha, quedando este último para uso exclusivo de la comunicación cliente-servidor.

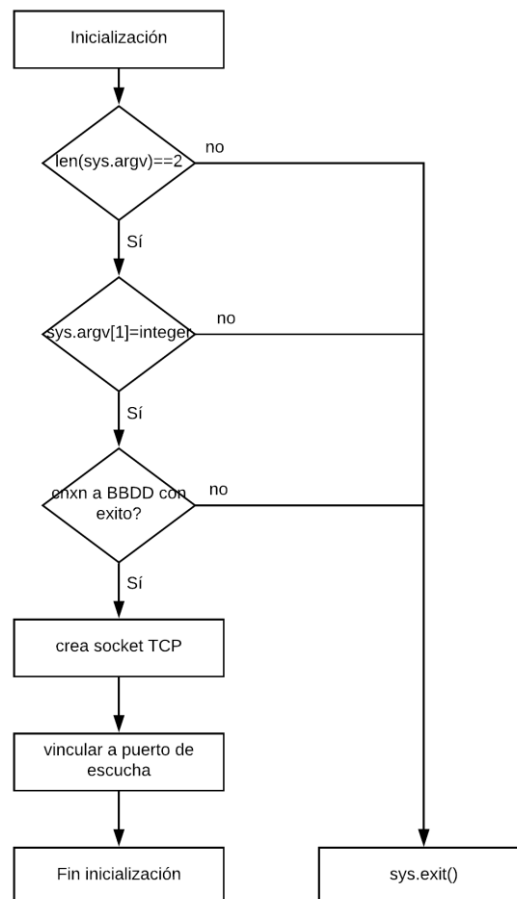


Figura 16 Diagrama de flujo de la inicialización del servidor

En el diagrama anterior pueden producirse errores no tratados explícitamente desde el código como el que salta al intentar vincular el socket a **un puerto ya 'ocupado' por otra aplicación** (excepción del tipo `socket.error`). Estos errores son tratados implícitamente por Python sin necesidad de incluir ninguna estructura del tipo `try/except`.

3.2.4 Funcionamiento

3.2.4.1 Espera de código RFID leído

Con la conexión establecida, el servidor envía el mensaje “**WAITING FOR RFID CODE**”, indicándole a la estación remota que puede proceder a enviar, en un segmento TCP, el código RFID leído cuando lo haya procesado. La función de Python de recepción de mensajes, una vez llamada, espera que en el buffer TCP haya datos listos para leer (o que la conexión se haya interrumpido).

3.2.4.2 Recepción y procesamiento de código RFID

Tras recibir el código leído, toca realizar la consulta a la BBDD para saber si consta en nuestro sistema como registrado. Esta tarea es llevada a cabo mediante una sentencia SQL del tipo SELECT. La sintaxis es la siguiente:

```
SELECT [COLUMNAS A DEVOLVER] FROM table_name WHERE [CONDICION]
```

Figura 17 Estructura genérica de una sentencia SQL de búsqueda en BBDD

La sentencia del tipo anterior toma tres grupos de parámetros:

- Columnas que devolver: En el caso de que no nos interese que la sentencia nos devuelva todas las columnas de las filas, solamente las que nos interesen.
- Nombre de la tabla: Ya que dentro de la BBDD puede haber distintas tablas de datos relacionadas entre sí.
- Condición/es: Restricciones que deben cumplir las filas de la tabla a seleccionar, tanto de igualdad (o desigualdad), como de mayor/menor en casos de valores numéricos. En caso de ser varias restricciones podrán usarse operadores lógicos (AND/OR...).

Dicho esto, es sencillo determinar la sentencia SQL concreta a la tarea que queremos realizar en la BBDD. Queremos saber si cierto valor se encuentra en alguna fila, por tanto

```
SELECT nombre,apellido,email,id FROM usersdb WHERE rfid='_rfid'
```

Figura 18 Sentencia SQL de búsqueda en BBDD de usuario

Siendo `_rfid` el valor recibido correspondiente al intento de autenticación desde el cliente. Debido a la configuración de la tabla (que se verá en su apartado correspondiente) **el máximo de filas que pueden cumplir esta condición es uno** (que correspondería con una consulta exitosa), y el mínimo, como es lógico, cero.

Para la transacción con la BBDD es necesaria una variable del tipo `cursor` y llamar al método `execute` de dicho tipo de variable. La llamada pone en marcha la consulta, almacenando el resultado en la propia variable de tipo `cursor` en forma de pila de datos.

Para acceder a cada fila individual de dicha pila de datos (estando formada cada fila por columnas correspondientes a cada dato solicitado en la sentencia SQL) **se usa el método `fetchone`** del tipo de variable `cursor`. En una aplicación de un propósito más general (donde número de coincidencias pueda ser más incierto, **accederíamos a todas las filas de la respuesta a través de una estructura en forma de bucle** (o iterativa) con `for` o `while`):

```
usuario = cursor.fetchone()
while (usuario is not None)
    #...
    #Procesamiento de datos recibidos
    #...
    usuario = cursor.fetchone()
```

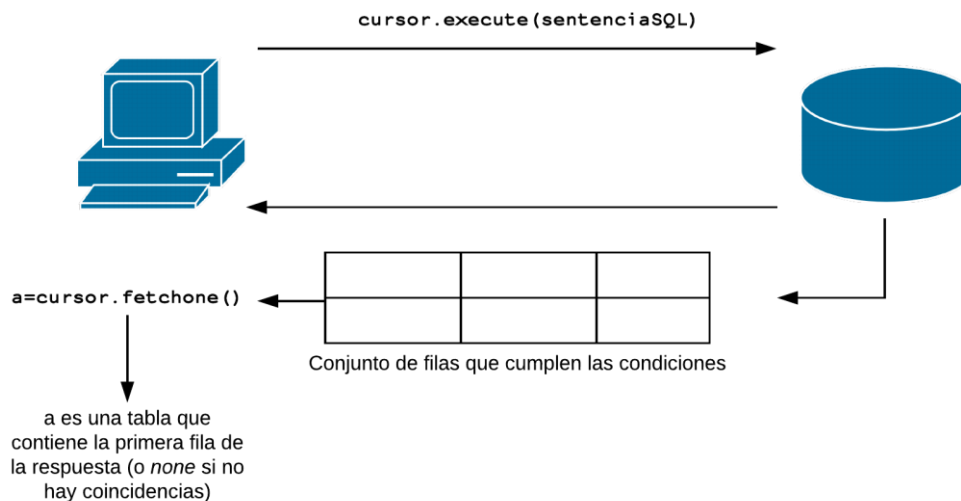


Figura 19 Secuencia de acciones en una consulta a la BBDD

Una vez realizada la consulta, tenemos que informar al cliente del resultado de esta. Dependiendo del resultado, la respuesta será

- **QUERY FAILED:** Si no hemos encontrado coincidencia (*matching* en inglés). Volveremos a esperar un nuevo código, y repetir la consulta con el nuevo.
- **QUERY SUCCESS:** Si la búsqueda ha encontrado resultado. Tendremos una tabla con cada uno de los campos pedidos en la sentencia SQL. También se envía el apellido del cliente, para representación en el LCD.

```
Tarjeta detectada, codigo RFID obtenido: 12-31-12-45-31
Usuario Registrado en la BBDD Remota

Datos del usuario identificado:
ID: 4
Nombre: Pepe
Apellido: Contreras
Correo electronico: pepe@hotmail.com
```

Ilustración 3-12 Consola del servidor cuando se intenta autenticar un usuario registrado en la BBDD

Como en otra tabla de cualquier otro tipo, los elementos son direccionables con un índice numérico, **estando ordenados según se haya especificado en la sentencia SQL de consulta**, siendo el índice cero el primero (a diferencia de otros lenguajes de programación donde el índice de la tabla comienza en uno).

El siguiente diagrama resume brevemente el proceso de autenticación en la BBDD.

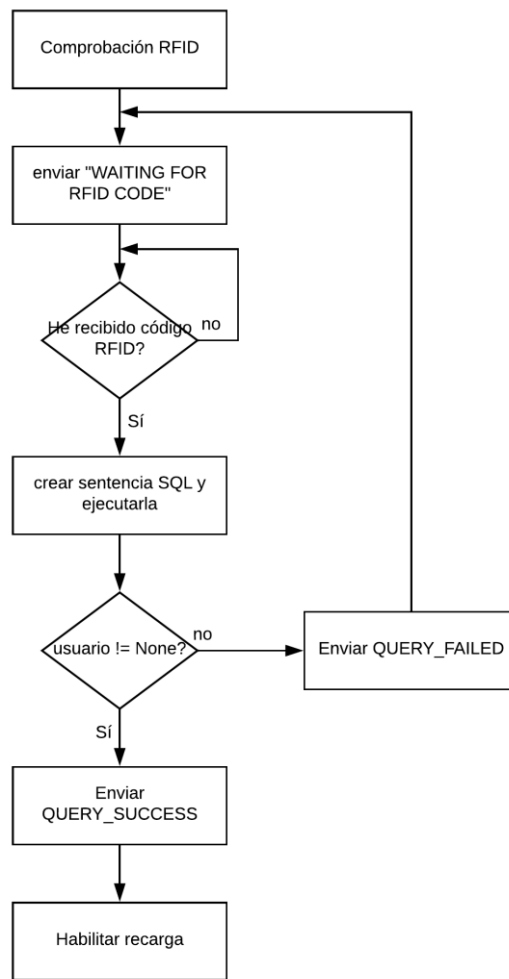


Figura 20 Diagrama de flujo de la autenticación

3.2.4.3 Seguimiento y cómputo del ciclo de recarga

Habiendo comprobado que, efectivamente, el usuario se encuentra registrado en la BBDD procederemos a realizar, en servidor, el cómputo de energía consumida de manera remota por el dispositivo enchufado. Como ya vimos en su apartado correspondiente, a modo de resumen:

1. El cliente cuando recibe el **“QUERY SUCCESS”** por parte del servidor **cierra el circuito del relé**, permitiendo el paso de corriente hacia el enchufe.
2. Hace ‘polling’ (es decir, vigila el valor) del pin conectado a la salida de pulsos del contador de consumo. **Cada vez que detecta un pulso nuevo, envía “PULSO”**.
3. Cuando **transcurre un determinado intervalo de tiempo** (configurable mediante código) desde el último pulso que contabilizó (o en su defecto, del inicio del cómputo), considera que, o bien, el dispositivo ha sido desconectado, o bien, ha completado su carga hasta cierto nivel. En ese caso envía **“FIN”**.

Volviendo ahora al lado del servidor, éste únicamente debe **de encargarse de contar cuantos “PULSO” recibe** (pues el resto del proceso es delegado al cliente), hasta que recibe “FIN”. Esta tarea es muy fácilmente implementable desde un bucle `while`, en el que la condición de iteración es que el dato leído del buffer de recepción TCP sea diferente a “FIN”. Adicionalmente, aunque no es estrictamente necesario, se verifica, antes de incrementar la variable contadora, que lo que ha recibido, sea precisamente “PULSO”. El total de la energía consumida, dado el número de pulsos, nos lo proporciona la relación pulsos/energía del medidor empleado:

$$\text{Energía consumida} = n_{\text{Pulsos}} * \text{Energía/Pulsos}_{\text{Contador}}$$

Según las especificaciones del convertidor, tras 1600 pulsos, se habrá consumido 1kWh, por tanto:

$$\frac{1 \text{ kWh}}{1600 \text{ pulsos}} = \frac{1000 \text{ Wh}}{1600 \text{ pulsos}} = \boxed{0.625 \frac{\text{Wh}}{\text{pulso}}}$$

Por tanto, la energía de un ciclo de recarga (expresada en Vatios hora) será de 0.625 por el número de pulsos contabilizados.

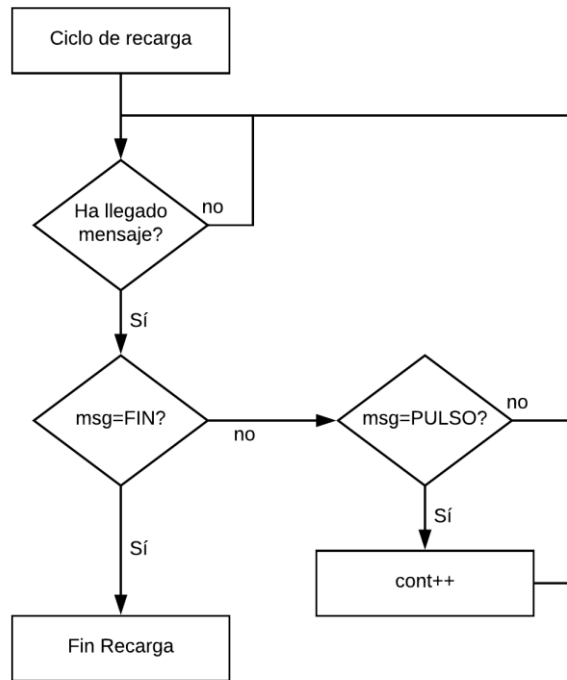


Figura 21 Diagrama de flujo del proceso de recarga

3.2.4.4 Registro de la recarga en la BBDD

Paralelamente a la tabla de clientes registrados del sistema, existe una tabla que **guarda un historial de todas las recargas** realizadas por todos los usuarios. Dicha tabla se relaciona con la de clientes por medio de la columna que contiene el identificador numérico del usuario **id_usuario**. Hasta ahora no hemos hablado mucho de esta columna porque es un dato sin más interés más allá del servidor.

Resumiendo, al terminar un ciclo de recarga hemos de guardar constancia de ello, o lo que es lo mismo, introducir una nueva entrada en la tabla correspondiente con la ayuda de la sentencia SQL de tipo INSERT, que tiene esta estructura:

```
INSERT INTO table_name [COLUMNAS] VALUES [VALOR DE CADA COLUMNA]
```

Figura 22 Estructura genérica de una sentencia SQL tipo INSERT

Desde un punto de vista estricto no es necesario (si vamos a darle valor a todas las columnas de la tabla, como es el caso que nos ocupa en este apartado) incluir el nombre de estas en la sentencia, sin embargo, es considerado una buena práctica hacerlo, pues reduciremos el riesgo de equivocarnos introduciendo valores en columnas equivocadas.

Los datos por introducir en la BBDD son:

- El **identificador numérico del usuario** autenticado, para relacionarlo con el resto de sus datos en la otra tabla.

- La **fecha y hora** en el momento que termina la recarga. Obtenidas a través de la librería de Python correspondiente (`datetime`).
- Un **identificador numérico del proceso de recarga actual**. Únicamente sirve para enumerar dentro de la BBDD. Sin interés más allá de este propósito.
- La **energía consumida en el presente proceso**. Calculada (estimada más bien) a través del número de pulsos medidos en el cliente remoto.

Habiendo aclarado todo lo anterior, la orden SQL para la tarea descrita tendría la siguiente forma:

```
INSERT INTO historial_recargas (num_recarga, date, id_usu, pot_consumida)
VALUES (A, B, C, D)
```

Figura 23 Sentencia SQL de inserción en BBDD

Hay una diferencia con respecto a la orden `SELECT` vista y explicada apartados atrás. Si únicamente ejecutamos la orden con el método `execute`, **no ocurrirá nada**, es decir, no estará la nueva entrada en la BBDD, como si no hubiéramos hecho nada.

La solución pasa por la llamada a una función adicional tras la correspondiente a `execute`. **Se trata del método `commit`** (del tipo de variable `cursor`). Su uso es obligatorio para cualquier operación en la BBDD que implique cambios en los datos almacenados en ella (como pueden ser, por ejemplo, sentencias del tipo `INSERT INTO`, o `UPDATE`). Formalmente, en las BBDD basadas en MySQL, al realizar operaciones de escritura, los nuevos datos son guardados provisionalmente en una memoria auxiliar, y es precisamente el método `commit` el que hace que los cambios sean permanentes.

3.2.4.5 Notificación por correo electrónico

El último paso del ciclo de recarga es notificarle al usuario, mediante correo electrónico, que se ha hecho una recarga a su nombre, amén de información adicional que pudiera ser de su interés.

Esta tarea, para abordarla mejor, puede dividirse en subtareas más elementales (en el código no se ejecutan de manera independiente y secuencial).

1. **Recogida de datos necesarios** a incluir en el correo.
2. **Composición del objeto tipo `email`**, con sus campos y su contenido (lo que se entiende como *cuerpo del mensaje*).
3. Diálogo previo al envío del correo con el servidor SMTP (autenticación).
4. Envío, propiamente dicho, del correo electrónico al usuario.

3.2.4.5.1 Recogida de datos y composición del cuerpo del mensaje

La recopilación de datos se refiere a realizar una búsqueda, en la tabla de historial de recargas, referente a **todas las recargas del usuario que consten en la BBDD**. Esta operación es muy similar a la de la búsqueda del usuario en la BBDD para saber si está autenticado, menos por una diferencia, que el número de resultados no está acotado (en apartados anteriores el hecho de saber con certeza que el número máximo de *matches* era uno simplificaba significativamente el código).

En este momento dicha suposición deja de ser válida, por lo que hemos de iterar indefinidamente hasta que, como ya se describió en su momento el método `cursor.fetchone` arroje como resultado `None`, indicándonos que ya hemos procesado todas las filas que cumplen con la condición especificada en la orden SQL.

Teniendo el identificador numérico del usuario autenticado es inmediata la obtención del historial completo, pues recordemos que al introducir en la BBDD una entrada correspondiente a un ciclo de recarga **también incluíamos el identificador del usuario en ella**. La orden SQL tendría la forma siguiente, siendo muy similar en estructura a la ya explicada de búsqueda de usuario:

```
SELECT date,pot_consumida FROM historial_recargas WHERE id_usu='_id'
```

Figura Sentencia SQL de búsqueda de historial completo

En el correo electrónico, al principio de este se le notifica al usuario del consumo energético de su recarga más reciente. Podemos aprovechar que tenemos almacenado en variables su nombre y apellidos (de la primera consulta) para dar la impresión de un trato más personalizado.

Llegados a este punto, toca ir completando secuencialmente el mensaje de correo con el resultado de la consulta, accediendo a cada campo con el índice cero y uno de cada fila individual, para ir formando el historial que aparecerá en el correo.

La estructura que construye secuencialmente el correo es (nótese que el índice de la tabla es debido al orden de las columnas en la consulta SQL):

```
while(dato is not None):
    fecha=dato[0]
    potencia=dato[1]
    str_msg=str_msg+fecha+'\t'+potencia+'\n'
    dato=cursor.fetchone()
```

Construcción secuencial de la tabla

```
Saludos Guillermo Roche.

Gracias por utilizar nuestro sistema electrico.

Acaba usted de realizar una recarga de 0.0Wh. A continuacion le adjuntamos el historial
de recargas de su cuenta que figura en nuestro sistema.
Hora y fecha          Consumo (Wh)
=====
15:33 31/07/2018     0.0
15:52 31/07/2018     0.0
15:59 31/07/2018     3.125
16:04 31/07/2018     0.0
05:46 08/08/2018     0.0
05:46 08/08/2018     0.625
05:47 08/08/2018     0.625
20:54 08/08/2018     0.0

Gracias por confiar en nuestro sistema de recarga remoto

La empresa
```

Ejemplo de correo electrónico que llega al usuario

3.2.4.5.2 Composición del objeto tipo 'email'

Con esto quedaría formado el cuerpo del mensaje en su totalidad, en un tipo de variable `String`, sin embargo, la librería trabaja con objetos tipo `email`, para lo cual tendremos que realizar la conversión oportuna. Para ello, la mencionada librería `email` pone a nuestra disposición la función `message_from_string`, que se ajusta perfectamente a lo que necesitamos, colocando además **la cadena de texto pasada como parámetro en el campo correspondiente al cuerpo del mensaje**.

Solo restaría ajustar los campos correspondientes a:

- Destinatario
- Remitente
- Asunto



Figura 24 Conversión de cadena de texto a objeto 'email'

Para modificar estos campos se usa una notación muy parecida a un acceso corriente a los campos de una tabla indexada mediante valores numéricos, excepto que los elementos tienen asociado un nombre en lugar de números.

También conviene recordar que disponemos de la dirección de correo del usuario almacenada en una variable desde el momento que se autenticó en el sistema, por lo que el proceso de asignación es muy sencillo, ya que tanto la dirección origen como el asunto serán los mismos siempre.

3.2.4.5.3 Autenticación en servidor SMTP y envío de correo

Una pregunta frecuente que podría hacerse el lector es acerca del **motivo de no realizar esta tarea al inicio de la ejecución del servidor**. La respuesta es sencilla, muchos servidores de correo tienen configurado un 'timeout' tras el cual, de no haber actividad por parte del cliente, cierran la sesión. Al ser un tiempo, en principio, aleatorio entre recarga y recarga, se opta por abrir y cerrar la sesión de correo con cada recarga individual.

La autenticación comienza con la creación del socket de conexión con el servidor SMTP, con el método `smtplib.SMTP`, que toma como parámetros la dirección IP del servidor y el puerto, siendo el estandarizado (que no por ello obligatorio), el número 587/TCP.

Tras la autenticación, se envía el saludo al servidor, pudiendo ser este comando HELO o EHLO según la funcionalidad que se desee (EHLO corresponde a ESMTP, o *Extended SMTP*, una versión mejorada de SMTP). El saludo se lleva a cabo con el método `ehlo()`.

Si se desea encriptar los mensajes (ya que los datos de acceso al correo se considera información delicada de cara a posibles ataques) se pueden usar los servicios de un protocolo de criptografía como TLS (*Transport Layer Security*). Este protocolo se sitúa en capa cuatro (capa de transporte) sobre TCP.

Una vez establecido un canal seguro, podemos enviarle nuestros datos de inicio de sesión al servidor, con el método `login()`.

Con el objeto email creado del apartado anterior, enviar el correo es tan sencillo como llamar a la función `sendmail()`. Cuando el correo ha sido enviado, el ciclo termina, y vuelve a enviar a la estación remota 'WAITING FOR RFID CODE' esperando una nueva autenticación.

3.2.5 Diagrama de flujo simplificado completo del servidor

Para concluir la sección describiendo al servidor se adjunta el siguiente esquema resumido sobre su funcionamiento.

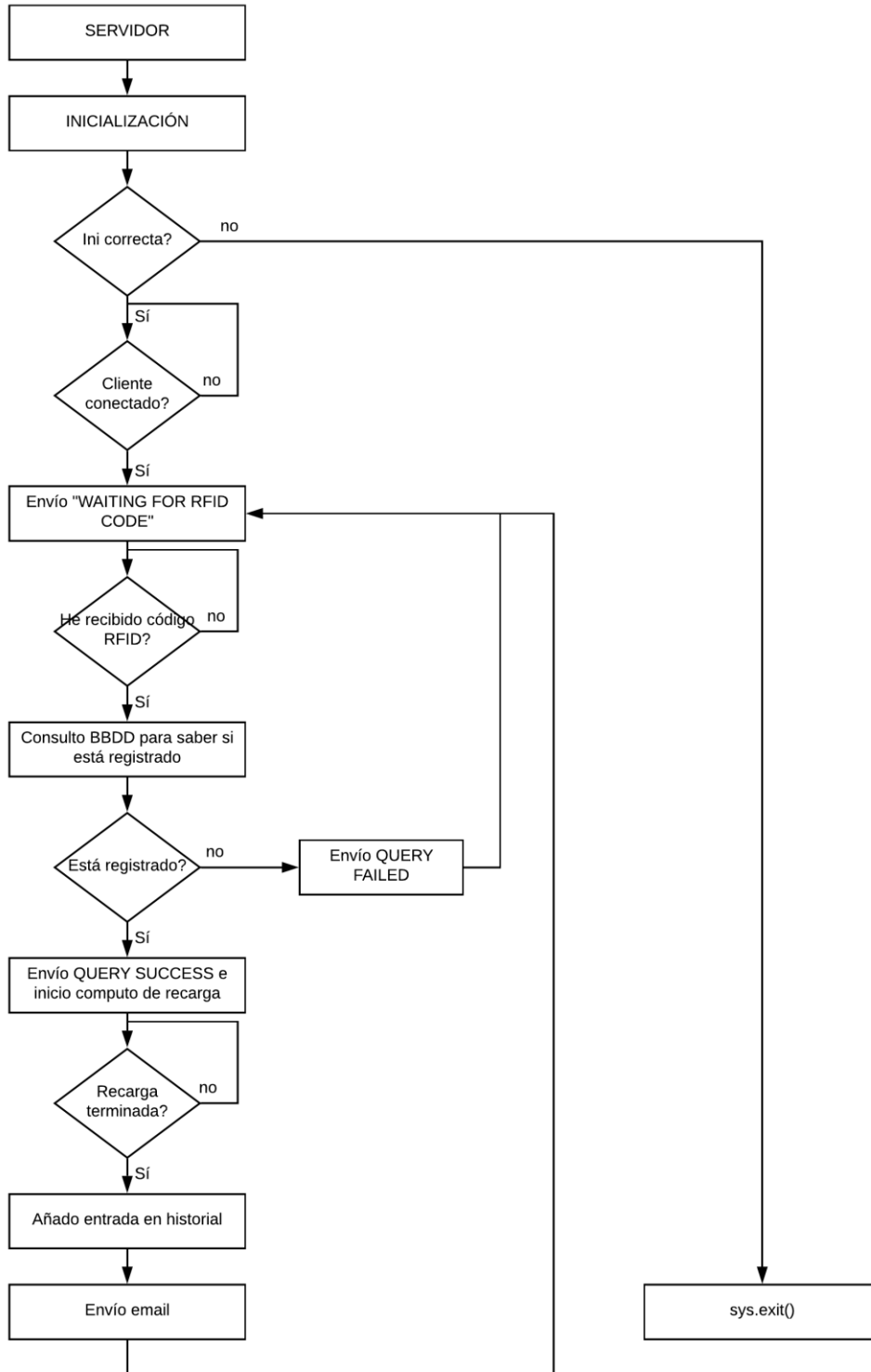


Figura 25 Diagrama de flujo completo del servidor

4 MONTAJE DEL SISTEMA

4.1 Software

Las herramientas software usadas en este proyecto son diferentes en servidor, cliente y BBDD.

4.1.1 Software en cliente (Arduino IDE)

Está basado en el entorno propio desarrollado por Arduino para sus placas. Tendremos que descargar el instalador de la página:

<https://www.arduino.cc/en/Main/Software>

Donde se podrá seleccionar el sistema operativo que estemos usando. En el O.S. (*Operative System*) Windows 10, además, **puede descargarse también por la tienda de Microsoft** de manera totalmente gratuita.

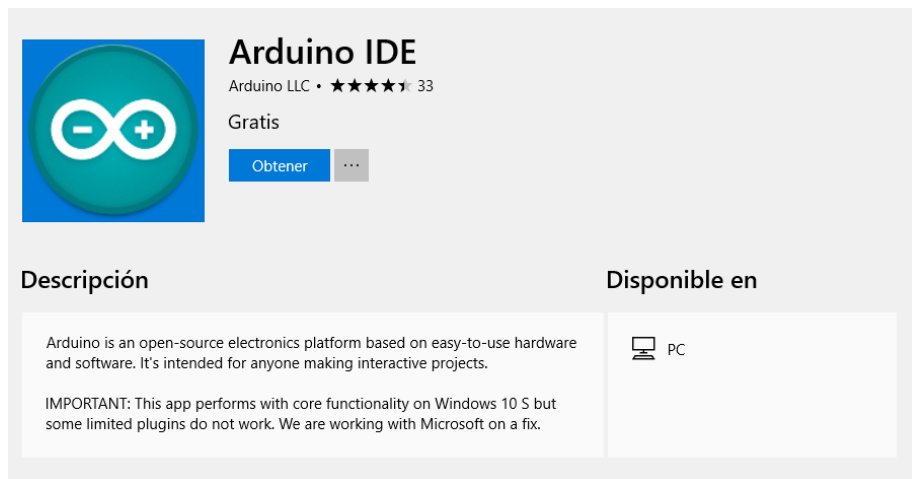


Ilustración 4-1 Arduino IDE disponible a través de la tienda de Microsoft

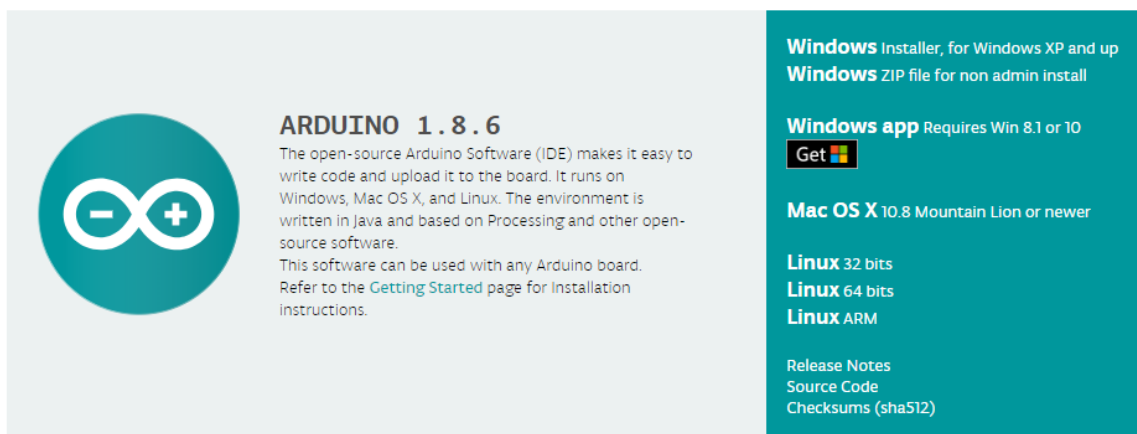


Ilustración 4-2 Arduino IDE en la web de Arduino

La instalación no tiene pérdida, y en unos sencillos pasos tendremos el entorno Arduino completamente operativo. El siguiente paso es la instalación de la librería RFID.

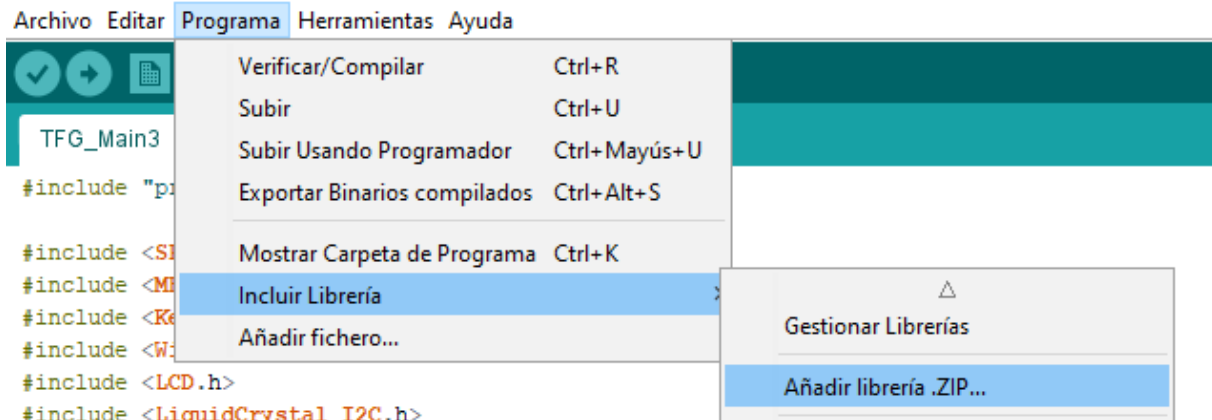


Ilustración 4-3 Inclusión de librerías externas en Arduino

Tras pinchar en la opción se nos abrirá una ventana de exploración de archivos donde le especificaremos al sistema la ruta al fichero “**RFID-master.zip**” para la instalación de este. Un proceso muy parecido es el que realizaremos para la instalación de la librería de manejo de LCD por medio del bus I2C (**LiquidCrystalI2C.zip**).

Para compilar únicamente el proyecto de Arduino podemos pulsar el botón de ‘verificar’, que tiene el símbolo de un tic.

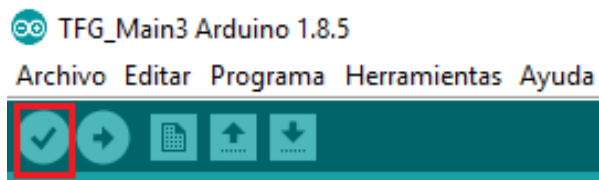


Ilustración 4-4 Opción de compilación del proyecto Arduino

Para poder cargar el programa en la tarjeta Arduino hará falta indicarle al IDE en qué puerto USB está conectada nuestra placa Arduino. Para ello acudimos al administrador de dispositivos de Windows.

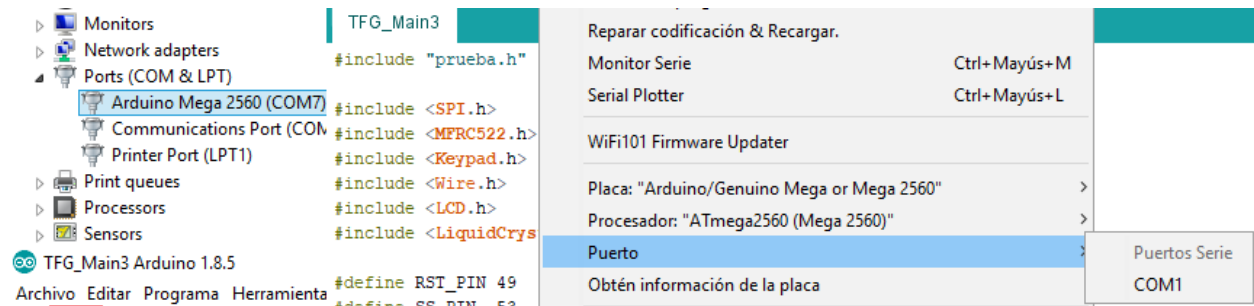


Ilustración 4-5 Configuración de puerto USB en el Arduino IDE

Para compilar y cargar el programa en la placa Arduino es necesario presionar el botón de ‘subir’

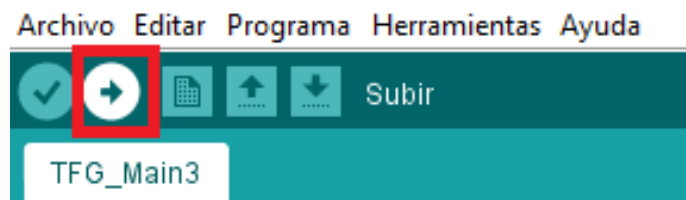


Ilustración 4-6 Carga de programa en tarjeta Arduino

4.1.2 Servidor

Para poder ejecutar el servidor necesitaremos instalar el intérprete de Python, en el OS que estemos usando. La URL de descarga es:

<https://www.python.org/downloads/>



Ilustración 4-7 Descarga de Python para Windows

Por defecto, en Windows el intérprete de scripts se ubica en la carpeta `C:\PythonXX`, donde XX representa la versión del intérprete. En sistemas operativos Linux no es necesario instalar el intérprete, pues viene de serie con el sistema.

Para ejecutar el script en Windows movemos el archivo fuente (archivo `TFGServer.py`) a la ubicación antes mencionada. Abrimos una ventana de consola, también llamada ‘símbolo del sistema’ y nos colocamos en el directorio raíz de la instalación de Python.

Por defecto Windows abre la ventana de comandos en el directorio del usuario (`C:\Users\nombre`) por lo que será necesario subir dos escalones en la jerarquía de ficheros para poder situarlo en el directorio correcto.

```
C:\Users\guill>cd..
C:\Users>cd..
C:\>cd Python27
C:\Python27>python TFGServer.py 3000
Escuchando
```

Ilustración 4-8 Recorrido hasta el directorio raíz de Python

Recordemos, pues, de apartados anteriores, que el comando para ejecutar el servidor es:

```
C:\Python27>python TFGServer.py [nPuerto]
```

En sistemas operativos Linux el comando es exactamente el mismo, con la diferencia procedimental que el archivo fuente no tiene por qué estar en el directorio raíz de Python (en la ventana de comandos tendremos que ir a la ubicación del código fuente), ya que el propio sistema operativo reconoce el comando, sin embargo, Windows llama al ejecutable `Python.exe`, por lo que deberá estar en la misma ruta.

4.1.3 Base de datos

Lo primero que necesitamos para implementar la BBDD es descargarnos el entorno MySQL, el cual puede obtenerse de:

<https://dev.mysql.com/downloads/>

New! MySQL Shell (GPL)

(Current Generally Available Release: 8.0.12)

The MySQL Shell is an interactive Javascript, Python, or SQL interface supporting development and administration for the MySQL Server and is a component of the MySQL Server.

[DOWNLOAD](#)**MySQL Workbench** (GPL)

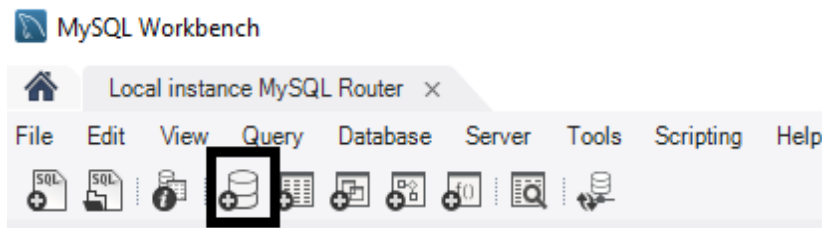
(Current Generally Available Release: 8.0.12)

MySQL Workbench is a next-generation visual database design application that can be used to efficiently design, manage and document database schemata. It is available as both, open source and commercial editions.

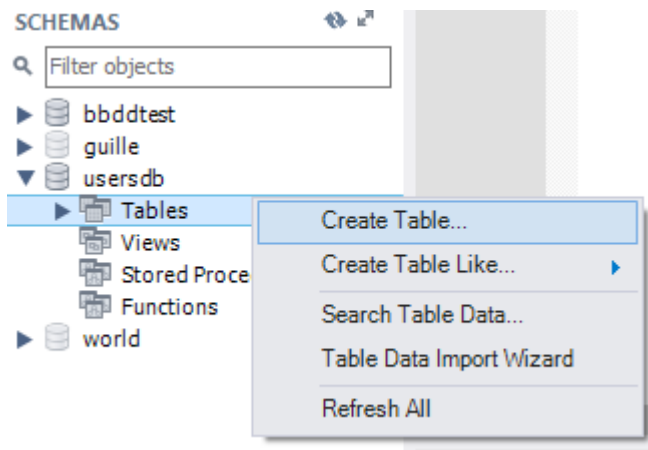
[DOWNLOAD](#)*Ilustración 4-9 Descarga de MySQL Workbench*

Durante la instalación se ajustarán parámetros clave del sistema gestor de base de datos (SGBD) como es el usuario y la clave de administrador.

Una vez la instalación haya finalizado, crearemos la BBDD donde estarán alojadas las tablas.

*Ilustración 4-10 Creación de BBDD en MySQL*

Especificaremos el nombre 'usersdb' (para que sea coherente con el código Python) y pulsaremos en 'Aplicar'. El siguiente paso es la creación de tablas dentro de la BBDD. En el menú 'schemas' de la parte izquierda de la ventana seleccionamos la base de datos, y dentro de ella, clic al botón izquierdo en 'tables'.

*Ilustración 4-11 Acceso al menú de creación de tablas*

Se abrirá el menú correspondiente, donde en la parte superior podremos elegir el nombre que queramos ponerle a la tabla y los nombres y tipos de variable de los datos que compondrán las columnas, además de las propiedades específicas de cada columna, por ejemplo, clave primaria, campo obligatorio... No está de más recordar que en SQL el tipo de variable String es conocido como Varchar(X) donde X es la longitud máxima de cadena aceptable para esa columna.

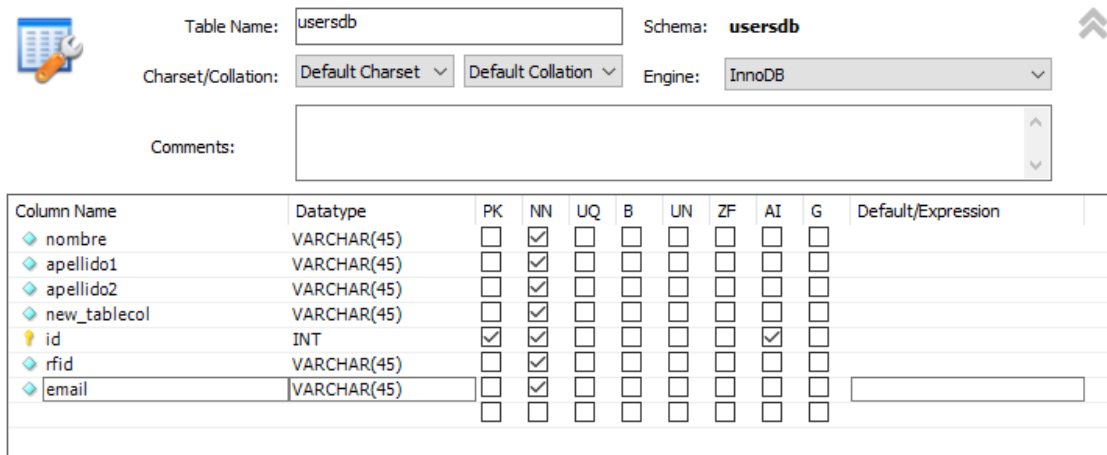


Ilustración 4-12 Ejemplo de creación de tabla

Cuando se pulsa en ‘Apply’ el entorno genera un script en lenguaje SQL que es la traducción de las acciones que hemos realizado en el entorno visual anterior. Simplemente pulsamos en ejecutar y ya estaría completada la creación de la tabla.

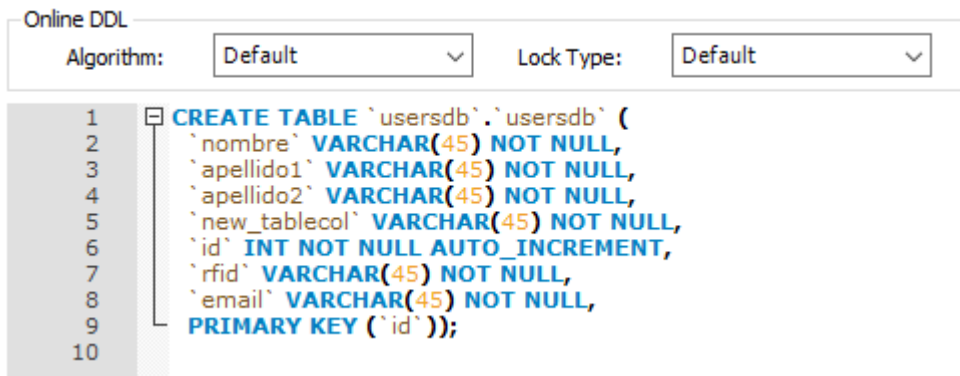


Ilustración 4-13 Script de creación de tabla generado automáticamente

De igual análoga se crearía la tabla para guardar el historial de recargas. Es tarea nuestra proveer de datos de clientes a la tabla de usuarios, para que puedan autenticarse en la estación remota. Para ello pinchamos con clic derecho en la tabla y elegimos ‘select rows’

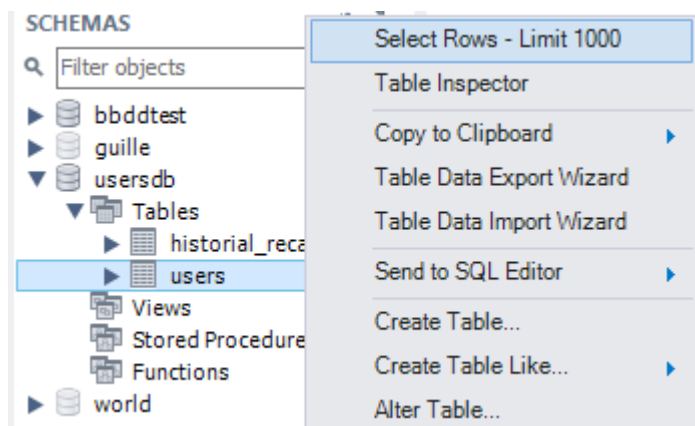


Ilustración 4-14 Visualización del contenido de la tabla

En la tabla mostrada existirá una fila con todos los valores a null. Pinchando en cada uno de los campos podremos darle el valor que queramos. Cuando hayamos terminado, pulsaremos en ‘apply’ y un nuevo script se generará automáticamente para escribir esas filas en la tabla.

| nombre | apellido1 | apellido2 | id | rfid | email |
|-----------|-----------|-----------|------|----------------|----------------------------|
| Guillermo | Roche | Arcas | 1 | 1-a4-3f-2e | quille_tenis93@hotmail.com |
| Jose | Perez | García | 2 | 11-22-33-44-55 | ioseperoar@gmail.com |
| Carlos | Roche | Arcas | 3 | eb-38-54-c3 | orafix993@gmail.com |
| Pepe | Contreras | Fernández | 4 | 12-31-12-45-31 | pepe@hotmail.com |
| Jesús | Pérez | Sanchez | 5 | 12-34-45-43 | jespersan@ejemplo.es |
| NULL | NULL | NULL | NULL | NULL | NULL |

Ilustración 4-15 Visualización del contenido de la tabla e inserción de nuevas filas

4.2 Hardware (estación remota)

En esta sección se profundizará sobre el montaje de la estación remota, pues el resto del proyecto no involucra configuración hardware. Puede dividirse en tres bloques funcionales

- Ajuste de la etapa de potencia (convertidor DC-DC) y rectificador
- Conexión del cuadro eléctrico y salidas.
- Conexión de la placa Arduino con los periféricos.

4.3.1 Montaje del cuadro eléctrico y relé

En esta parte se va a conectar los siguientes componentes:

- Interruptor general.
- Interruptor diferencial
- Relé.
- Medidor de consumo.
- Enchufe tipo hembra.

Adicionalmente necesitaremos una clavija de corriente tipo macho y conductor para la conexión a la red eléctrica, que irá conectada en serie con el interruptor general:



Ilustración 4-16 Conexión entre la red eléctrica y el interruptor general

También, en serie, tras el interruptor general, va el interruptor diferencial:



Ilustración 4-17 Conexión entre interruptor general y diferencial

Lo último que nos quedaría sería la interconexión entre el interruptor diferencial, el medidor de consumo



Ilustración 4-18 Conexión completa del cuadro eléctrico

Las conexiones que realizar son las siguientes:

- Enroscar el cobre de los extremos de dos cables y conectarlo a uno de los dos terminales libres del interruptor diferencial.
- Uno de los cables sueltos irá a terminal denotado como '4' del medidor de consumo.
- El otro al terminal 'NC' del relé/
- El terminal común del relé (COM) irá a uno de los terminales del enchufe.
- El terminal que queda libre del interruptor diferencial va conectado al terminal número 1 del medidor de consumo.
- El terminal restante (el número 3) del medidor de consumo va al terminal libre del enchufe.

Adicionalmente, colocaremos dos cables a la salida de pulsos del medidor de consumo, para conectarlos a la placa Arduino.

4.3.2 Alimentación

Como ya especificamos en el desglose de componentes que la alimentación de periféricos corría a cargo de un sistema específicamente pensado para ello, ya que no era aconsejable usar las salidas de tensión del Arduino si hay muchos periféricos. Dicho sistema de potencia se compone de:

- Fuente de alimentación: Va conectada a la red eléctrica y a su salida proporciona una tensión continua de entre 11 y 12 V
- Convertidor DC-DC ajustable manualmente, mediante potenciómetro, variando la tensión de salida entre 0V y la entrada.

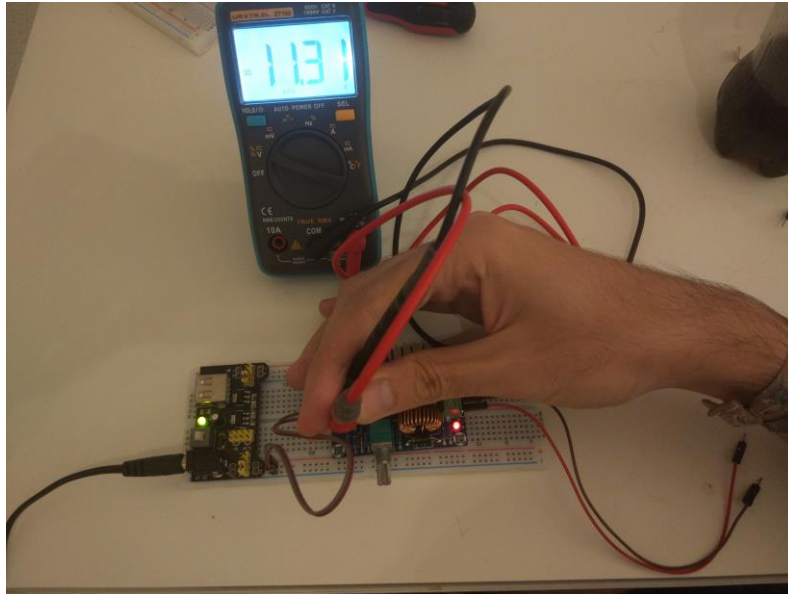


Ilustración 4-19 Medición de la tensión de salida de la fuente de alimentación

La conexión entre ambos componentes es en serie, teniendo especial cuidado con la polaridad (es decir, uniendo terminales positivos y negativos).

Con los elementos ya enlazados podemos realizar el ajuste del transformador. El proceso consiste en ir girando el potenciómetro hasta obtener una tensión (razonablemente) igual a 5V.

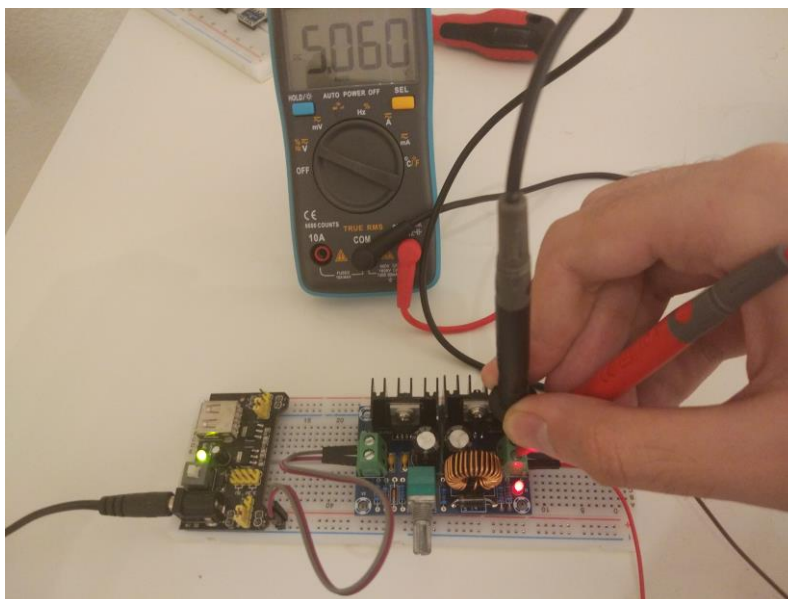


Ilustración 4-20 Ajuste de la tensión de salida del transformador a 5V

Cuando hayamos completado el ajuste, el montaje del sistema de alimentación estará terminado

4.3.3 Sistema microcontrolador Arduino

En este apartado se especificará como conectar cada periférico al sistema microcontrolador de Arduino. Como ya se especificó en apartados anteriores los periféricos son los siguientes:

- Display LCD y expensor de puertos mediante conexión I2C
- Sensor RFID mediante SPI.
- Tarjeta de red mediante UART.
- Relé y medidor de consumo mediante pines de propósito general GPIO.

4.3.3.1 Display LCD y expensor de pines

No hay mucho que comentar aquí. La placa Arduino tiene pines dedicados exclusivamente al protocolo I2C, por lo que únicamente habría que conectar ambos pines de datos y reloj del expensor de pines (SDA/SCK) a sus homónimos de la placa. No es necesaria la disposición de resistencias pull-up externas pues vienen implementadas en la placa. También habría que conectar tierra y alimentación al transformador.

4.3.3.2 Tarjeta de red

La conexión de la tarjeta de red GPRS no es tan trivial como la del display. Cuenta con ocho pines, de los cuales solo usaremos cinco. Pueden clasificarse en tres tipos:

- Alimentación y tierra, denotados como '5V' y GND. Van conectados al transformador.
- Comunicación. Al ser full dúplex requiere una línea de transmisión y otra de recepción, denotadas como T y R en la tarjeta. Van conectadas a los pines RX1 y TX1 de la placa Arduino.
 - Esta configuración no es la única posible, podrían usarse los pares de pines TX2/RX2 y TX3/RX3, sin embargo, requeriría cambios en el código, básicamente sustituir '`serial1`' por '`serial2`' o '`serial3`' según corresponda.
- Control: Un solo pin, denotado como 'K'. A nivel alto mantiene la tarjeta apagada, mientras que a nivel bajo habilita el funcionamiento. Como, por simplicidad, no nos interesa en este proyecto alternar ambos estados, simplemente cortocircuitamos este pin a tierra para que la tarjeta esté siempre funcionando al disponer de alimentación.

4.3.3.3 Relé y medidor de consumo

El relé únicamente necesita de un pin de propósito general para poder ser controlado. La elección es totalmente libre, eso sí, el pin elegido debe concordar con el almacenado en la constante de programa '`PIN_PULSOS`' definida en código.

En cuanto a la conexión con el medidor de consumo, hemos de cortocircuitar ambas tierras. La salida positiva debe ser ir conectada a un pin que soporte ser asociado a una interrupción hardware, que en las placas Arduino Mega son 2,3,18,20 y 21.

4.3.3.4 Sensor RFID

La librería de Arduino empleada para el manejo del periférico usa cinco pines fijos y dos variables (configurables por el usuario).

- En los pines fijos se ubican los dos de alimentación (a 3.3V), conectados a la salida de tensión de Arduino, y los tres para la comunicación. El pin denotado como SCK va al pin 52, MOSI al 51, y MISO al 50.
- Los pines variables pueden ser asignados a cualquier GPIO disponible. El pin conectado a SS tiene que coincidir con la constante `SS_PIN` y el RST, a su vez, con `RST_PIN`

REFERENCIAS

Arduino. (s.f.). *Arduino Reference*. Obtenido de <https://www.arduino.cc/reference/en/>

Nokia Networks Oy. (s.f.). Obtenido de CiteSeerX:
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.697.7011&rep=rep1&type=pdf>

Oracle Corporation. (s.f.). *MySQL Reference Manual*. Obtenido de <https://dev.mysql.com/doc/refman/8.0/en/>

Python Software Foundation. (s.f.). Obtenido de SMTP Library Reference:
<https://docs.python.org/2/library/smtplib.html>

Ross, J. F. (2013). *Computer Networking: A top-down approach*. Pearson.

ANEXO A: FUNDAMENTOS DE REDES GPRS

La red GPRS da servicio de transporte de tráfico de paquetes desde los terminales móviles, hasta otras redes de paquetes diferentes, pudiendo ser la interred (lo más común) u otra privada.

Esta tecnología surgió posteriormente a GSM (*Global System for Mobile Telecommunications*), y en lugar de reemplazarla completamente, se diseñó para funcionar en paralelo a ella, ya que sus servicios eran complementarios, amén de no inutilizar los dispositivos anteriores. Los requisitos que debe cumplir una red GPRS son los siguientes.

- Tiene que usar en la medida de lo posible **toda la infraestructura** (ya instalada previamente) de GSM con el menor número de modificaciones.
- Como existe la posibilidad de que los usuarios participen en más de una sesión de intercambio de datos, debe poder soportar **más de una conexión** por conmutación de paquetes.
- Debe poder ofrecer **diferentes estándares de calidad** (QoS) en función de lo que los usuarios quieran contratar, y del dinero que estén dispuestos a gastar.
- Debe poder ser compatible, y **funcionar de manera paralela**, de igual forma que lo hace con GSM, con las posteriores generaciones de telefonía móvil (actualmente 3G y 4G), para no provocar incompatibilidad con los terminales antiguos.
- Debe ser capaz de soportar tanto las conexiones punto a punto, como punto a multipunto.
- Debe ser capaz de dar servicio al usuario para poder conectarse a redes externas de manera segura.

1. Estructura y elementos de la red GPRS

En un escenario muy simplificado, la red del operador con servicio GPRS tendría una estructura similar a la que se muestra a continuación:

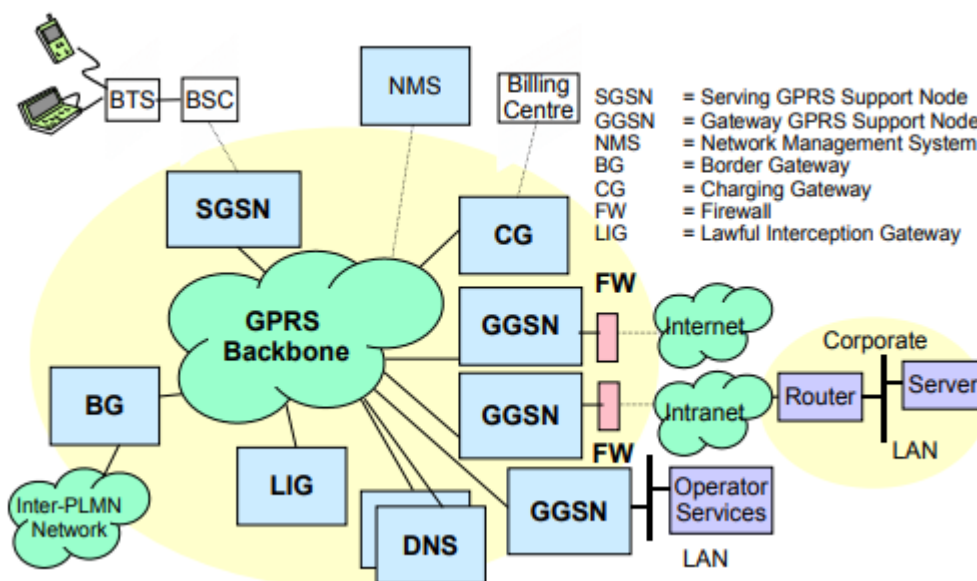


Figura 26 Estructura genérica de una red GPRS

- **SGSN:** Es el elemento más importante de la red, equivalente al MSC (Mobile Switching Centre) de

GSM. Realiza funciones de autenticación, pasarela, gestión de la movilidad del cliente y optimización de la carga útil del paquete.

- **GGSN:** Es una pasarela (Gateway) entre la red GPRS del operador, y otras, pudiendo ser Internet, o alguna intranet. Entre sus funciones se encuentra, entre otras, encaminar los paquetes a la SGSN en la que se encuentra el abonado destino de dicha información, asigna direcciones IP a los terminales móviles, con la posible ayuda de un servidor DHCP...
- **BG:** Es una pasarela que da acceso directo a otras PLMN (redes públicas móviles) dado que es más seguro transportar esta información de una red a otra directamente que
- **CG (Charging Gateway):** Realiza funciones de tarificación basadas en el volumen de datos, calidad de servicio ofrecido (QoS)... para luego comunicarse con el centro de tarificación (Billing Centre)
- **Firewalls (FW):** Importantes para la seguridad frente a ataques, tanto desde dentro de la red como desde Internet. Suelen estar configurados para desechar todos los paquetes IP que no procedan de una sesión iniciada por un abonado de la red GPRS, es decir, toda comunicación con el exterior debe ser iniciada por el equipo que se encuentra en la red GPRS.
- **Servidores DNS:** No difieren en nada con respecto a cualquiera de internet. Su función es la de traducir nombres (p. ej., en el proyecto que se describe en este documento, de 'airtelwap.es') a direcciones IP.

2. Transferencia de paquetes entre nodos de red (GSN) GPRS

Los paquetes se transmiten por el núcleo de la red en lo que se denomina 'contenedores'. Este transporte de datos es completamente transparente para el usuario, de esta manera, tiene la impresión de estar conectado a un host con un router de intermediario.

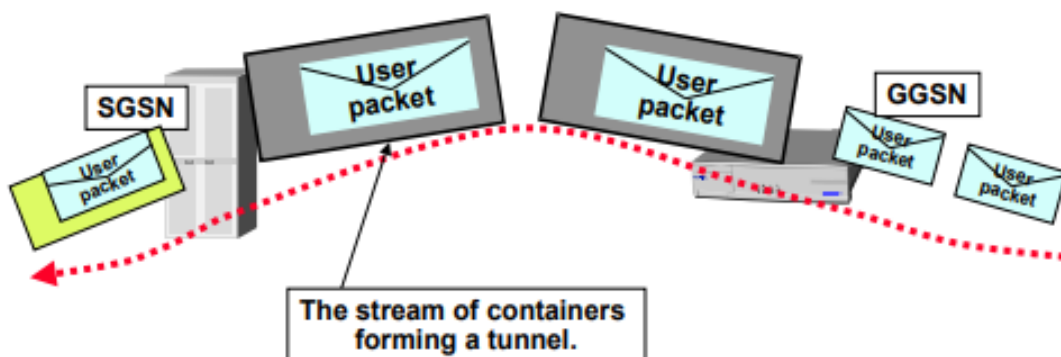


Figura 27 Tunnelling en GPRS

En el campo de la telemática, este procedimiento recibe el nombre de 'tunnelling', que en el tipo de redes que nos ocupa, se encarga el protocolo de 'tunnelling' GPRS, conocido por sus siglas GTP (*GPRS Tunnelling Protocol*).

En el interior de la red GPRS, los paquetes GTP van encapsulados dentro de paquetes IP, y dentro de estos paquetes GTP puede, a su vez, ir encapsulados los paquetes TCP/IP que conforman el tráfico entrante en la red. De esta manera, se establece un 'doble nivel IP', el de más "bajo nivel OSI", correspondiente a la red GPRS (que es IP, a fin de cuentas), y el de más alto nivel, que corresponde a Internet. Debido a esta arquitectura de protocolos, las direcciones IP pueden repetirse en ambas redes.

Este hecho lo observamos en su momento al ejecutar el comando AT+XIIC? en el módem GPRS, éste nos devolvía una dirección de red en un rango privado, típicamente 10.A.B.C.

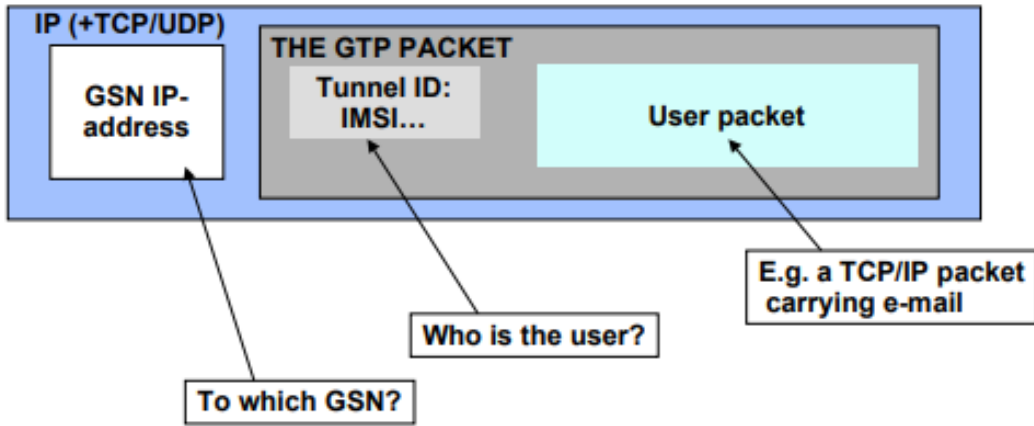


Figura 28 Encapsulación de mensajes en redes GPRS

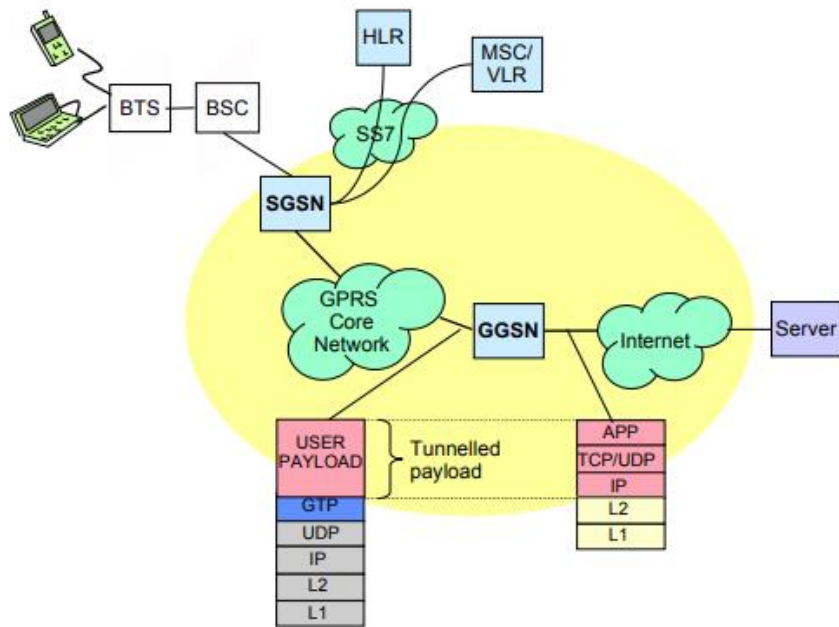


Figura 29 Torres de protocolos en la frontera de GPRS

En las imágenes anteriores se puede observar gráficamente el concepto de encapsulamiento en redes GPRS, cómo las capas superiores OSI (es decir, IP y a todo lo que da servicio), se encapsula como carga útil dentro de la torre de protocolos propia de la torre GPRS. Es la pasarela GGSN (frontera entre la red GPRS e Internet) la que se encarga de hacer esta labor de “traducción”.

ANEXO B: PROTOCOLO NAT Y CONFIGURACIÓN DEL ROUTER

1. Breve introducción teórica a NAT y su funcionamiento

El protocolo NAT surge como **solución**, llamada por algunos expertos en el tema “solución temporal”, **al problema originado por agotamiento progresivo de direcciones IPv4 libres**, debido al crecimiento exponencial de dispositivos conectados a Internet en la última década. Dicho de otra manera, **llegó un momento el cual treinta y dos bits no eran suficientes para direccionar a todos los equipos conectados a Internet**. Se basa en la asignación de direcciones IP ‘especiales’, conocidas como ‘privadas’, a los equipos (hosts) conectados a una red local (LAN).

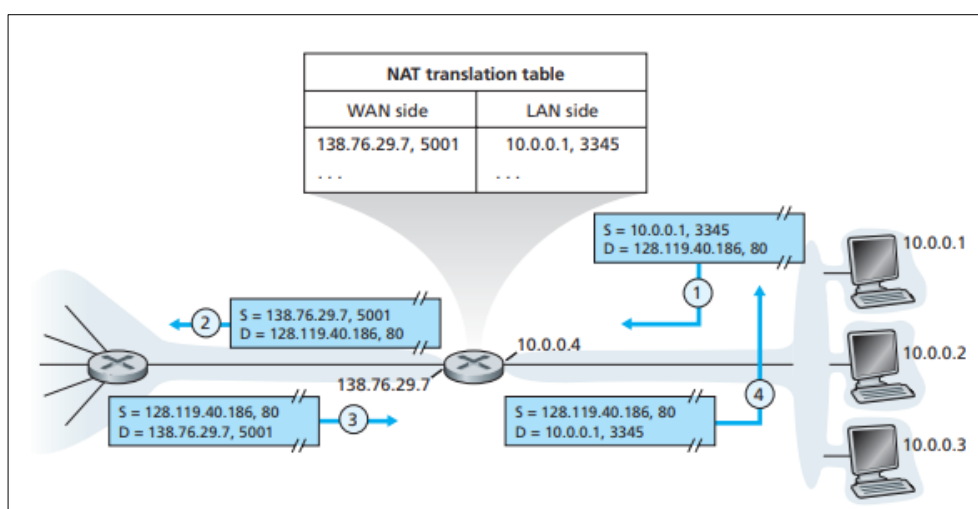


Figura 30 Funcionamiento básico de NAT

Dichas direcciones privadas sólo tienen validez en el ámbito local de la red (dicho de otro modo, no más allá de la interfaz LAN del router). Para que las aplicaciones distribuidas residentes en los equipos, identificadas por la IP del equipo y puerto (TCP/UDP) puedan establecer comunicación con el resto de Internet, necesitan una ‘nueva identidad’ válida en el ámbito de la interred.

Para ello, el router NAT **asigna de manera unívoca a cada pareja de IP y puerto de la red local un número de puerto de su interfaz WAN** (que sí que tiene una IP ‘pública’). De este modo, todo el tráfico de red externo que le llegue al router con destino a un puerto suyo determinado, se reenvía a la red local cambiando la dirección IP y puerto destino del paquete por el que corresponda, atendiendo a la tabla de traducción NAT interna del dispositivo.

Siguiendo el ejemplo de la figura, cualquier paquete que lleve como puerto destino el 5001, será redirigido a la IP de la red local 10.0.0.1 y puerto 3345.

Este proceso de asignación se realiza, por defecto, **de manera automática** (el propio router decide qué puerto “darnos”) para comodidad de los usuarios, pero también existe la posibilidad de hacerlo nosotros mismos, casi siempre cuando queremos instalar un servidor en nuestra red local (ya que nos interesa reservarnos un determinado puerto del router y que ese sea siempre el mismo).

De otra manera, el router cada vez nos asignaría un puerto distinto para el servidor, provocando que los clientes no sepan a dónde enviar sus peticiones de conexión.

2. Relación con el proyecto

Si ejecutamos el comando '>ipconfig' en la consola de comandos de Windows obtendremos, entre otra información, la dirección IP (privada en este caso) del equipo:

```
Microsoft Windows [Versión 10.0.16299.431]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\guill>ipconfig

Configuración IP de Windows

Adaptador de Ethernet Ethernet:

    Sufijo DNS específico para la conexión. . . : home
    Vínculo: dirección IPv6 local. . . . . : fe80::31f7:c0df:2d8b:ee0f%4
    Dirección IPv4. . . . . : 192.168.1.51
    Máscara de subred. . . . . : 255.255.255.0
    Puerta de enlace predeterminada. . . . . : 192.168.1.1
```

Ilustración 0-1 Configuración de red en la consola de Windows

Como hemos explicado en el apartado anterior, no podemos limitarnos a poner esta dirección (192.168.1.51) en el comando de establecimiento de conexión TCP del chip GSM (AT+TCPSETUP...) ya que sólo funcionaría en el supuesto caso de que el servidor se encontrara en el mismo ámbito local que el cliente (supuesto imposible ya que el cliente se sirve de la red GPRS para su acceso a Internet).

La solución, por tanto, consiste en la configuración de la tabla NAT del router para asegurarnos que todo el tráfico que vaya dirigido a cierto puerto de su dirección pública se redirija al proceso servidor de nuestro equipo. La elección del número en la interfaz externa del router es, en principio, a libre elección.

El proceso de configuración que se presenta a continuación es **para los routers del fabricante CBN**, aunque es fácilmente extrapolable a cualquier otro. Ante las dudas, consulte el manual de su fabricante.

1. En un navegador, accedemos a la dirección IP asignada la interfaz interna (LAN) del router, que coincide con lo que en '>ipconfig' aparece como **puerta de enlace predeterminada**. En nuestro caso particular, se trata de '192.168.1.1'.
2. Introducimos la contraseña correspondiente al usuario administrador del router.
3. Accedemos, en el menú superior, a 'Router>Avanzada>Reenvío'



Ilustración 0-2 Acceso a la configuración de reenvío del router

Tras lo cual, accedemos a la configuración NAT del router (únicamente se muestran las filas creadas manualmente).

| REENVÍO | | | | | | | |
|--------------------------------------|----------------|--------------|----------------|--------------|-----------|-------------------------------------|--------------------------|
| Dirección IP externa: 217.216.238.13 | | | | | | | |
| Reenvío de puertos | | | | | | | |
| Addr IP local | Externo | | Interno | | Protocolo | Activado | Eliminar |
| | Puerto inicial | Puerto final | Puerto inicial | Puerto final | | | |
| 192.168.1.8 | 12000 | 12000 | 3000 | 3000 | Ambos | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| 192.168.1.8 | 15000 | 15000 | 5000 | 5000 | TCP | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| 192.168.1.51 | 20000 | 20000 | 2000 | 2000 | Ambos | <input checked="" type="checkbox"/> | <input type="checkbox"/> |

Ilustración 0-3 Configuración de la tabla NAT manual/estática del router

En ella, además, se **muestra la dirección IP que el ISP contratado ha asignado a la interfaz externa del router**, concretamente, '217.216.238.13'. En la tabla aparecen tres columnas que son las que nos interesan. Estas son: 'Addr IP local', 'Externo' e 'Interno'.

Por ejemplo, la primera fila de la tabla nos indica que todo el tráfico dirigido (tanto en el protocolo TCP como UDP) a **217.216.238.13:12000** será, redirigido al **puerto 3000** del host de la LAN cuya dirección sea **192.168.1.8**.

El proceso servidor escucha en el puerto 8000 (del equipo, no del router), por simplicidad, decidimos que **el router también tenga asignado externamente ese puerto para la aplicación**. Para ello pulsamos en 'Agregar' y rellenamos los campos con los valores correspondientes.

| REENVÍO | | | | | | |
|--------------------------------------|----------------|--------------|----------------|--------------|-----------|-------------------------------------|
| Dirección IP externa: 217.216.238.13 | | | | | | |
| Addr IP local | Externo | | Interno | | Protocolo | Activado |
| | Puerto inicial | Puerto final | Puerto inicial | Puerto final | | |
| 192.168.1.51 | 8000 | 8000 | 8000 | 8000 | Ambos ▾ | <input checked="" type="checkbox"/> |

Ilustración 0-4 Creación de nueva fila estática en la tabla NAT del router

Tras pulsar en 'Agregar' ya tendremos todo configurado correctamente.

Por tanto, para poder establecer correctamente la conexión TCP entre el proceso servidor alojado en un equipo, que a su vez está dentro de una red LAN, con un router NAT, tendremos que poner, en el comando correspondiente, la IP externa del router, junto con el puerto 8000.

ANEXO C: CÓDIGO PYTHON DEL SERVIDOR

```

# Importando librerias de socket y Mysql
import sys
import time
import socket
import mysql.connector
from mysql.connector.errors import Error
import datetime
import email
import smtplib

impToWh = 0.625;
npulsos = 0;
num_recarga = 0;
#1kWh=1600imp -> 1000Wh=1600imp -> 1 imp = 0.625 Wh
datoRFID=''
datoRecibido=''

if (len(sys.argv)!=2):          #Numero de argumentos incorrecto
    print '****ERROR****'
    print 'Uso del script:'
    print '>python TFGServer.py + (Numero de puerto TCP de escucha)'
    sys.exit()

TCP_IP = ''
try:
    TCP_PORT = int(sys.argv[1])          #Puerto proporcionado por usuario en
    linea de comandos
except ValueError:
    print("Error en el numero de puerto, debe estar formado unicamente por
    cifras")
    sys.exit()
BUFFER_SIZE = 1024
try:
    cnx = mysql.connector.connect(user='root',
    password='root',host='127.3.0.1', database='usersddb')
except mysql.connector.errors.ProgrammingError:
    print("Error conectando a BBDD, la BBDD no existe o la combinacion
    usuario/pass es incorrecta")
    sys.exit()
except UnicodeDecodeError:
    print("Error conectando a BBDD, revise la direccion IP")
    sys.exit()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
print 'Escuchando'
s.listen(5)
conn, addr = s.accept()
print 'Cliente conectado, IP Addr:', addr
time.sleep(2);
while (datoRFID != '0'):
    conn.send('WAITING FOR RFID CODE')
    print 'Esperando que el usuario acerque RFID'
    time.sleep(1);
    #Recibo codigo RFID por parte del cliente
    datoRFID = conn.recv(BUFFER_SIZE)
    print '\n*****\n'

```

```

print 'Tarjeta detectada, codigo RFID obtenido: ', datoRFID
#Sentencia SQL de consulta a la BBDD
query = ("SELECT nombre,apellido1,rfid,email,id FROM users WHERE
        rfid='"+datoRFID+"'")
cursor = cnx.cursor()
cursor.execute(query)
#Comprobacion de la existencia de un usuario con ese ID en la BBDD
usuario=cursor.fetchone()
if(usuario is not None):                                #Usuario Registrado
    time.sleep(1)
    conn.send('QUERY SUCCESS')
    time.sleep(2)
    nombre=usuario[0]
    apellido=usuario[1]
    correo=usuario[3]
    id = usuario[4]
print 'Usuario Registrado en la BBDD Remota\n\n'
print 'Datos del usuario identificado:'
print 'ID: ', id
print 'Nombre:\t\t', nombre
print 'Apellido:\t', apellido
print 'Correo electronico:', correo, '\n\n\n'
#conn.send('N:'+nombre)
conn.send('A:'+apellido)
print 'El usuario esta recargando'
while (datoRecibido!= 'FIN') :
    datoRecibido=conn.recv(BUFFER_SIZE)
    if (datoRecibido == "PULSO"):
        npulsos=npulsos + 1
print 'Se han detectado ', npulsos, 'pulsos. Resultando en una
        energia de ', impToWh*npulsos, 'Wh'
    now = datetime.datetime.now()
    sql = "INSERT INTO historial_recargas (num_recarga, date, id_usu,
        pot_consumida) VALUES (%s, %s, %s, %s)"
    val = (str(num_recarga), now.strftime("%H:%M %d/%m/%Y"), str(id),
        str(impToWh*npulsos))
    cursor.execute(sql, val)
    cnx.commit()

    str_msg=""
    sql = ("SELECT date,pot_consumida FROM historial_recargas WHERE
id_usu='"+str(id)+"'")
    cursor.execute(sql)
    dato=cursor.fetchone()
    str_msg+="Saludos "+nombre+" "+apellido+".\n\nGracias por utilizar
nuestro sistema electrico.\n\n"
    str_msg+="Acaba usted de realizar una recarga de " +
str(impToWh*npulsos) + "Wh."
    str_msg+=" A continuacion le adjuntamos el historial de recargas de
su cuenta que figura en nuestro sistema\n"
    str_msg+="Fecha/Hora\t\t\tConsumo (Wh)\n"
    str_msg+="=====\n"
while(dato is not None):
    fecha=dato[0]
    potencia=dato[1]
    str_msg=str_msg+fecha+'\t'+potencia+'\n'
    dato=cursor.fetchone()
    str_msg+="\n\nGracias por confiar en nuestro sistema de recarga
remoto\n\nLa empresa"
    #print str_msg
    msg = email.message_from_string(str_msg)
    msg['From'] = "CUENTA"

```

```
msg['To'] = correo
msg['Subject'] = "Recarga"

s = smtplib.SMTP("smtp.gmail.com",587)
s.ehlo() # Hostname to send for this command defaults to the fully
qualified domain name of the local host.
s.starttls() #Puts connection to SMTP server in TLS mode
s.ehlo()
s.login([cuenta], [PASS])

s.sendmail(correo, correo, msg.as_string())
s.quit()

datoRecibido=''
npulsos=0;
else:                                     #Usuario no registrado
print 'No existe ningun usuario con el codigo RFID especificado'
conn.send('QUERY FAILED')
time.sleep(1)

conn.close()
```


ANEXO D: CÓDIGO ARDUINO DEL CLIENTE

TFGMain.ino

```

#include <Wire.h>
#include "TFG_RFID.h"
#include "TFG_NIC.h"
#include "TFG_connError.h"

#include <SPI.h>
#include <MFRC522.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

#define VEL 9600
String serialBuffer;
String cadena;

String ACKSalidaTCP = "";

boolean TCPTX_FLAG = false;

//LiquidCrystal_I2C          lcd(I2C_ADDR,2, 1, 0, 4, 5, 6, 7);
void setup() {
  Serial1.begin(VEL);
  Serial.begin(VEL);
  SPI.begin(); //Iniciamos el Bus SPI
  mfrc522.PCD_Init(); // Iniciamos el MFRC522

  lcd.begin(16,2); // Inicializar el display con 16 caracteres 2 lineas
  lcd.setBacklightPin(3,POSITIVE);
  lcd.setBacklight(HIGH);
  lcd.print("INICIALIZANDO");
  gestionNIC=compruebaPIN;
  gestionRFID=reposoRFID;
  gestionError=reposoTCP;

  pinMode(CONTROL_PIN,OUTPUT);
  digitalWrite(CONTROL_PIN,HIGH);
}

void loop() {
  (*gestionNIC)();
  (*gestionRFID)();
}

void serialEvent1(void)
{
  serialBuffer=Serial1.readString();
  Serial.print(serialBuffer);
  if (serialBuffer.indexOf("+TCPSETUP") > 0) {
    TCPCONN_FLAG=true;
    bufferCnxnTCP=serialBuffer;
  }
  if (serialBuffer.indexOf("+TCPSEND") > 0) {

```

```

    TCPTX_FLAG=true;
    ACKSalidaTCP=serialBuffer;
  }
  if (serialBuffer.indexOf("+TCPRECV") > 0) {
    TCPRX_FLAG=true;
    TCPBufferRX=serialBuffer;
  }
  if (serialBuffer.indexOf("+XIIC") > 0) {
    NETSTAT_FLAG=true;
    netStatBuffer=serialBuffer;
  }
  if (serialBuffer.indexOf("+CPIN") > 0) {
    PINCHECK_FLAG=true;
    pinCheckBuffer=serialBuffer;
  }

  if (serialBuffer.indexOf("+TCPCLOSE:0,Link Closed") >= 0) {
    TCPERROR_FLAG=true;
    Serial.println("Error en la conexion. Se ha caido");
  }
}

```

TFG_NIC.h

```

#define CONTROL_PIN 47
#define PIN_PULSOS 2
#define UMBRAL_PULSO 80
#define SEGENTREPULSOS 10
#define puls_to_wh 0.625
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

boolean TCPERROR_FLAG = false;
boolean TCPCONN_FLAG = false;
boolean NETSTAT_FLAG = false;
boolean PINCHECK_FLAG = false;
boolean TCPRX_FLAG = false;

boolean NEWPULSE_FLAG = false;
boolean FINRECH_FLAG = false;

String bufferCnxnTCP = "";
String netStatBuffer = "";
String pinCheckBuffer = "";
String TCPBufferRX = "";

String nombreUsuario = "";

float lcd_wh = 0;

#define LINEABLANCO " "
#define I2C_ADDR 0x27
int nPulsos = 0;
long periodoAct = 0;
long periodoAnt = 0;
long periodo = 0;
LiquidCrystal_I2C lcd(I2C_ADDR,2, 1, 0, 4, 5, 6, 7);
void (*gestionNIC)(void);
int msSinPulsos = 0;
String IP_Server="217.216.227.7"; //IP del Servidor Remoto
String Puerto_server="12000";

```

```

void cuentaPulsos(void);
//=====
void configuraAPN(void);
void compruebaRed(void);
void esperaRespPIN(void);
void esperaRespuestaXIIC(void);
void connToServer(void);
void esperaRespServer(void);
void esperaIndServidor(void);
void enviaCodigoAlServidor(void);
void esperaRespRFID(void);
void esperaNombreRFID(void);
void esperaPulsos(void);
//=====
void espera() {
}

void compruebaPIN() {
  Serial1.println("ATE0");
  delay(1000);
  Serial1.println("AT+CPIN?");
  gestionNIC=esperaRespPIN;
}

void esperaRespPIN() {
  if(PINCHECK_FLAG == true) {
    PINCHECK_FLAG=false;
    if(pinCheckBuffer.indexOf("+CPIN: READY") >= 0) {
      Serial.println("SIM desbloqueada");
      gestionNIC=configuraAPN;
    }
    else {
      Serial.println("Error, la tarjeta SIM no está desbloqueada");
      gestionNIC=espera;
    }
  }
}

void configuraAPN() {
  delay(1000);
  Serial1.println("AT+CGDCONT=1,\"IP\", \"movistar.es\");
  delay(1000);
  Serial1.println("AT+XISP=0");
  delay(1000);
  Serial1.println("AT+XIIC=1");
  delay(2000);

  gestionNIC=compruebaRed;
}

void compruebaRed() {
  lcd.setCursor ( 0, 1 );
  lcd.print("RED MOVIL ");
  Serial1.println("AT+XIIC?");
  gestionNIC = esperaRespuestaXIIC;
}

void esperaRespuestaXIIC() {
  if(NETSTAT_FLAG == true) {
    NETSTAT_FLAG = false;
    if(netStatBuffer.indexOf("0.0.0.0") >= 0 |
netStatBuffer.indexOf("ERROR") >= 0) {

```

```

Serial.println("Conexión Fallida a la red");
lcd.print("FAIL");
gestionNIC=espera;
}
else {
Serial.println("Comprobacion exitosa de la conectividad");
lcd.print("OK");
gestionNIC=connToServer;
}
}
}

void connToServer(void) {
Serial1.println("AT+TCPSETUP=0," + IP_Server + "," + Puerto_server);
gestionNIC=esperaRespServer;
}

void esperaRespServer(void) {
if(TCPCONN_FLAG == true) {
lcd.clear();
lcd.home();
lcd.print("TCP CONNECTION");
lcd.setCursor(0,1);
TCPCONN_FLAG = false;
if (bufferCnxnTCP.indexOf("+TCPSETUP:0,OK") >= 0) { //Conexion
resstablecida con éxito
lcd.print("OK");
Serial.println("Conexion establecida con exito al servidor");
gestionNIC=esperaIndServidor;
}
else {
lcd.print("FAILED");
Serial.println("Conexión fallida al servidor");
gestionNIC=espera;
}
}
}

void esperaIndServidor (void) {
if(TCPRX_FLAG == true) {
TCPRX_FLAG = false;
if(TCPBufferRX.indexOf("WAITING FOR RFID CODE") >= 0) {
lcd.clear();
lcd.home();
lcd.print("Acerque RFID");
Serial.println("Esperando codigo RFID");
RFID_FLAG = true;
gestionNIC=enviaCodigoAlServidor;
}
}
}

void enviaCodigoAlServidor () {
if(RFID_completed_read == true & TCPERROR_FLAG == false) {
RFID_completed_read=false;
Serial1.println("AT+TCPSEND=0,"+ String(RFID_salida.length()));
//Serial.println("AT+TCPSEND=0,"+ String(RFID_salida.length()));
//Serial.println("#" + RFID_salida + "#");
delay(1000);
Serial1.print(RFID_salida);
Serial1.write(0x0D);
lcd.setCursor(0,1);
}
}
}

```

```

    lcd.print("Esp. Resp. Serv");
    gestionNIC = esperaRespRFID;
  }
}

void esperaRespRFID (void) {
  if(TCPRX_FLAG == true) {
    lcd.clear();
    lcd.home();
    lcd.print("Usuario");
    lcd.setCursor(0,1);
    TCPRX_FLAG = false;
    if(TCPBufferRX.indexOf("QUERY SUCCESS") > 0) {
      Serial.println("El usuario está registrado en la BBDD");
      lcd.print("registrado");
      gestionNIC=esperaNOMBRERFID;
    }
    else if(TCPBufferRX.indexOf("QUERY FAILED") > 0) {
      Serial.println("No se encontró al usuario en la BBDD");
      lcd.print("no registrado");
      gestionNIC=esperaIndServidor;
    }
  }
}

void esperaNombreRFID (void) {
  if(TCPRX_FLAG == true) {
    TCPRX_FLAG = false;
    lcd.clear();
    lcd.home();
    lcd.print("Bienv sr/sra");
    nombreUsuario =
TCPBufferRX.substring(TCPBufferRX.indexOf("A:")+2,TCPBufferRX.length()-2);
    lcd.setCursor(0,1);
    lcd.print(nombreUsuario);
    attachInterrupt(digitalPinToInterrupt(PIN_PULSOS), cuentaPulsos, RISING);
    msSinPulsos = 0;
    digitalWrite(CONTROL_PIN, LOW);
    periodoAct = millis();
    gestionNIC=esperaPulsos;
  }
}

void esperaPulsos (void) {
  if (NEWPULSE_FLAG == true) {
    if(TCPERROR_FLAG == false) {
      NEWPULSE_FLAG = false;
      Serial1.println("AT+TCPSSEND=0,5");
      delay(1000);
      Serial1.print("PULSO");
      Serial1.write(0x0D);
      lcd_Wh += puls_to_Wh;
      lcd.clear();
      lcd.home();
      lcd.print("Energia cons.");
      lcd.setCursor(0,1);
      lcd.print(String(lcd_Wh));
    }
    else {
      FINRECH_FLAG = true;
    }
  }
}
if (FINRECH_FLAG == true) {

```

```

    FINRECH_FLAG=false;
    Serial.println(msSinPulsos);
    detachInterrupt(digitalPinToInterrupt(PIN_PULSOS));
    Serial.println("FIN RECARGA");
    digitalWrite(CONTROL_PIN, HIGH);
    if(TCPERROR_FLAG == false) {
        Serial1.println("AT+TCPSSEND=0,3");
        delay(1000);
        Serial1.print("FIN");
        Serial1.write(0x0D);
    }
    gestionNIC=esperaIndServidor;
}
}

void cuentaPulsos(void) {
    periodoAnt=periodoAct;
    periodoAct=millis();
    periodo=periodoAct-periodoAnt;
    if(periodo > 80) {
        msSinPulsos=0;
        NEWPULSE_FLAG = true;
    }
    else {
        msSinPulsos += periodo;
        if(msSinPulsos > 10000) {
            FINRECH_FLAG=true;
        }
    }
}
}

```

TFG_RFID.h

```

#include <SPI.h>
#include <MFRC522.h>
#include <Wire.h>
#define RST_PIN 49 //Pin 9 para el reset del RC522
#define SS_PIN 53 //Pin 10 para el SS (SDA) del RC522
MFRC522 mfr522(SS_PIN, RST_PIN); //Creamos el objeto para el RC522
String RFID_salida;
char RFID_code[4][3];
int k=0;
void (*gestionRFID)(void);
boolean RFID_FLAG = false;
boolean RFID_completed_read = false;
void leeTarjetaRFID();

void reposoRFID()
{
    if ( mfr522.PICC_IsNewCardPresent() & RFID_FLAG == true)
    {
        RFID_FLAG=false;
        gestionRFID=leeTarjetaRFID;
        RFID_salida="";
    }
}

void leeTarjetaRFID()
{
    if ( mfr522.PICC_ReadCardSerial()
        {

```

```
RFID_completed_read = true;
Serial.println("Nueva Tarjeta detectada!");
for (byte i = 0; i < mfrc522.uid.size; i++) {
    itoa(mfrc522.uid.uidByte[i],RFID_code[i],16);
}
for(k=0;k<4;k++)
{
    RFID_salida.concat(RFID_code[k]);
    if(k!=3)
        RFID_salida.concat("-");
}
Serial.println("Codigo RFID leido: " + RFID_salida);
mfrc522.PICC_HaltA();
gestionRFID = reposoRFID;
}
```