Proyecto WJSN

Wireless Jarduino-Sensor Network

Descripción de la implementación de la red y sus nodos

SMSE

índice

1. Objetivos del documento

Los objetivos de este documento son tres:

- Describir el modo en el que se han programado los protocolos descritos en el documento adjunto "EspecificaciónProtocolos.pdf" en la biblioteca WJSN.h (adjunta a este documento.
- Describir la construcción, programación de los distintos nodos y el despliegue de una red inalámbrica que se ha llevado a cabo a modo de ejemplo.
- Aclarar algunas limitaciones de la implementación y posibles mejoras o ampliaciones.

2. Biblioteca WJSN.h

Introducción

La biblioteca WJSN.h consta de dos archivos. En WJSN.h se declaran las tres clases que conforman la biblioteca, se declaran las variables y métodos que conforman cada una de las clases, se definen las constantes necesarias mediante la orden "#define" y se incluyen las bibliotecas necesarias mediante "#include". En el fichero WJSN.cpp se implementan todos los métodos declarados en WJSN.h.

Las tres clases que se definen son:

- La clase "mensaje" que servirá para interpretar una cadena de bytes como un mensaje de los diferentes protocolos descritos en el documento "EspecificaciónProtocolos.pdf".
- La clase "nodoPasarela" que contiene los métodos y variables necesarias para implementar y gestionar fácilmente un nodo pasarela.
- La clase "nodoSensor" que servirá para implementar y gestionar fácilmente un nodo sensor, conteniendo los métodos y variables necesarias para ello.

Clase mensaje

La clase mensaje contiene una serie de métodos (get*() y set*()) para extraer los diferentes campos y para cambiar el valor de los campos. Además, tiene dos métodos (enviar() y enviarS()) que sirven para enviar el mensaje contenido por una instancia de la clase por radio o por el puerto serie, respectivamente.

Además, contiene un puntero a uint8_t (_mensaje) que apuntará a una tabla de tipo uint8_t donde estará almacenado el mensaje. Este puntero será asignado de forma dinámica, por lo que también hay una variable de tipo uint8_t que almacena el tamaño de la cadena apuntada por _mensaje. Esta variable se llama _tam.

A parte de los métodos ya descritos, existen dos métodos más con funciones particulares. Está definido el método "imprimirCPG (uint8_t* tam)" y "setMsg (uint8_t* datos, uint8_t tam".

- imprimirCPG (uint8_t* tam): Este método devuelve una tabla (uint8_t*) reservada dinámicamente con el contenido CPG encapsulado en el mensaje precedido de un octeto (uint8_t) que contiene la dirección origen del mensaje. Sirve a la hora de encapsular los mensajes PGD sobre UDP en la pasarela. La longitud de la tabla devuelta la guarda en la dirección apuntada por tam.
- setMsg (uint8_t* buff, uint8_t tam): Establece los valores _mensaje y tam del objeto mensaje a los recibidos buff y tam. Es para instanciar un objeto mensaje de forma dinámica.

Clase nodoPasarela

Esta clase almacena los parámetros relativos a una pasarela en sus variables y contiene los métodos necesarios para hacer envíos descendentes hacia la red y comunicarse con un cliente que desee recibir mediciones desde la red. Contiene las variables:

- IPAddress _ipAsociado: Guardará el valor de la IP del cliente que se haya asociado con la pasarela para recibir mediciones.
- Unsigned int _puertoAsociado: Guardará el número de puerto UDP que el cliente usa para comunicarse con la pasarela.
- IPAddress _ip: Almacena la dirección IP de la propia pasarela.
- Unsigned int _puerto: Almacena el número de puerto UDP que se abrirá para recibir y enviar mensajes a los posibles clientes.
- Uint8_t estado: Variable que define el estado en el que se encuentra la pasarela: ASOCIADO o ESPERA.
- Uint8_t _tablaRD[][2]: Tabla que almacenará las direcciones de los nodos que se han asociado con la pasarela.
- Uint8_t _tamTablaRD: Variable que indica el número de elementos guardados en _tablaRD.
- Uint8_t _id: Número de identificación de la pasarela en la red de sensores.
- Uint8_t _incremento_temporal: Tiempo, en segundos, que se espera antes de actualizar el contador de segundos y se comprueba si se han recibido mensajes.
- Uint8_t _contSegundos: cuenta los segundos que han pasado desde la última vez que se actualizaron los contadores de los distintos protocolos.
- Uint8_t _contAnuncio: cuenta los minutos que faltan hasta que se envíe un nuevo mensaje de tipo anuncio, si la pasarela está en estado ASOCIADO.
- EthernetUDP _UDP: Objeto UDP que se empleará para enviar y recibir mensajes UDP hacia el cliente.

Contiene los métodos:

- inicializarNP(uint8_t id, IPAddress ip, byte* mac, unsigned int puerto, uint8_t incrementoTemporal): Establece los valores que definen la pasarela para comenzar el funcionamiento.
- void actualizarNP(): Actualiza los contadores necesarios, envía los mensajes necesarios según los protocolos y procesa los mensajes recibidos en el último periodo de espera. Después esperará tantos segundos como indique el valor de _incrementoTemporal.
- void procesarCEPER(mensaje Msg): Procesar el mensaje CEPER encapsulado en Msg.
- void anunciar(): envía un mensaje anunciar, anunciándose a sí mismo.
- void ack(mensaje Msg): Responde al mensaje Msg con un mensaje de tipo ACK.
- int comprobarRD(uint8_t ID, uint8_t pasarela): Comprueba que el nodo con identificador ID está en la Tabla de Reenvío Descendente.
- void procesarPER(mensaje Msg): Procesa el mensaje PER contenido en Msg según el protocolo PER.
- void aceptarPER(mensaje Msg): Llama a las funciones de los protocolos de capa superior al PER según se indique en el campo PROT de Msg.
- void reenvioD(mensaje Msg): Envía el mensaje Msg a los nodos asociados con la pasarela.

- void procesarPGD(mensaje Msg): Reenvía apropiadamente el contenido PGD de Msg hacia el cliente.

Clase nodoSensor

Esta clase contiene las variables necesarias para almacenar los parámetros de funcionamiento de un nodo sensor junto con los métodos apropiados para ejecutar los protocolos implicados en su correcto funcionamiento (PGD, PER y CEPER). Para ello comparte las siguientes variables con la clase "nodoPasarela":

_estado, _tablaRD, _tamTablaRD, _id, _incrementoTemporal, _contSegundos y _contAnuncio.

Además de estas, contiene las variables:

- uint8_t _tablaRA[]: Contendrá las direcciones de las pasarelas que el nodo ha descubierto, los siguientes saltos para alcanzarlas y el número de saltos hasta ellas.
- uint8_t _tamTablaRA: Número de pasarelas descubiertas.
- uint8_t _contDescubrir: Minutos restantes para el envío de un mensaje de tipo DESCUBRIR.
- uint8_t _contMedicion: Minutos restantes para realizar una medición.
- uint8_t* _bandera, _tam, _datos: Punteros que conforman el interfaz con el código del usuario de la biblioteca. _bandera apuntará hacia una celda que marcará si hay una nueva medida cargada. _tam apuntará a una celda que indicará el número de medidas que se han realizado. _datos apuntará a una tabla que contendrá *(_tam) medias y sus respectivos identificadores de tópico.

En cuanto a métodos, comparte con la clase "nodoPasarela" los siguientes:

procesarCEPER(), anunciar(), descubrir(), comprobarRD(), ack(), procesarPER(), aceptarPER() y reenvioD().

Además de estos métodos, contendrá:

- void inicializarNS(uint8_t id, uint8_t incrementoTemporal, uint8_t* bandera, uint8_t* tam, uint8_t* datos): Inicializa las variables que definen el comportamiento del nodo a los valores indicados para comenzar el funcionamiento del nodo.
- void actualizarNS(): Actualiza los contadores necesarios, envía los mensajes necesarios según los protocolos y procesa los mensajes recibidos en el último periodo de espera. Después esperará tantos segundos como indique el valor de _incrementoTemporal.
- void envioA(mensaje Msg): Envía el mensaje Msg hacia todas las pasarelas descubiertas por el nodo.
- void reenvioA(mensaje Msg): Reenvía el mensaje Msg hacia la pasarela apropiada.
- void asociar(mensaje Msg): Responde al mensaje Msg (que debe ser de tipo ANUNCIAR) con un mensaje de tipo ASOCIAR.
- int comprobarRA(uint8_t IDpasarela): Comprueba si la pasarela de id IDpasarela está incluida en la Tabla de Reenvío Ascendente.
- void descubrir(): Envía un mensaje de tipo DESCUBRIR.
- int getEstado(): puede ser usado por el usuario de la biblioteca para conocer el estado del nodo en tiempo de ejecución.

3. Código de la pasarela

Fichero WJSN_pasarela.ino

Atendiendo a las descripciones de las tres clases hechas en el apartado anterior de este documento, presentamos el código empleado en el nodo de tipo pasarela (ver Código 1).

```
/*
 *  Nombre: WJSN_sensor.ino
 *  Autor: Pablo Linares Serrano
 *
 *  Descripción: Implementación de un nodo sensor WJSN empleando la
 *  biblioteca WJSN.h
 */
// inclusión de bibliotecas. La biblioteca VW está incluida en WJSN.h
#include <WJSN.h>

// Definición de parámetros
#define ID 1
#define INCREMENTO 1

// Declaración del objeto nodo WJSN
nodoPasarela pasarela;
// Declaración de la mac e ip
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
IPAddress ip(169, 254, 1, 177);
unsigned int puertoloc = 5027;

void setup() {
  pasarela.inicializarNP(ID, ip, mac, puertoloc, INCREMENTO);
}

void loop() {
  pasarela.actualizarNP();
}
```

Código 1: Fichero WJSN_sensor.ino.

Como se ve en la Código 1, establecemos los valores de la dirección IP, el número de puerto y el ID de forma estática, inicializándolos en el objeto "nodoPasarela" en la función "setup()". Para ello nos valemos del método "inicializarNP" descrito anteriormente.

En la función "loop()" ejecutamos el método "actualizarNP()", que actúa según el estado del nodo pasarela, como ya se ha indicado antes.

Fichero WJSN.cpp

El código de los métodos "inicializarNP()" y "actualizarNP()" pueden verse en el Código 2 y el Código 3 respectivamente. Éste está implementado en el fichero WJSN.cpp, perteneciente a la biblioteca WJSN.

```
void nodoPasarela::inicializarNP(uint8_t id, IPAddress ip, byte* mac,
unsigned int puerto, uint8_t incrementoTemporal){
       // Inicializa un nodo pasarela, estableciendo los valores apropiados
en sus campos en tiempo de ejecución.
       Serial.begin(9600);

    // inicialización de los pines de los led
       pinMode(LED_RX_PIN, OUTPUT);
       pinMode(LED_TX_PIN, OUTPUT);
       pinMode(LED_ASOCIADO_PIN, OUTPUT);
       pinMode(LED_ESPERA_PIN, OUTPUT);

       // asignación de valores a variables
       _incrementoTemporal = incrementoTemporal;
       _id = id;
```

```
        _estado = ESPERA;
        _tamTablaRD = 0;
        _contSegundos = 0;
        _contAnuncio = T_ANUNCIO;

        // ponemos a cero las variables propias de los nodos tipo sensor:
        _ip = ip;
        _puerto = puerto;

    // inicializamos el puerto ethernet
        Ethernet.begin(mac, _ip);

        //inicializamos el puerto UDP
        _UDP.begin(puerto);
}
```

Código 2: método "inicializarNP()".

```
void nodoPasarela::actualizarNP(){
        // Actualiza todos los contadores necesarios,
        // Comprueba si se han recibido mensajes y responde de ser necesario
        // Comprueba si es necesario enviar nuevos mensajes
        uint8_t buf[VW_MAX_MESSAGE_LEN];
    uint8_t buflen;
    uint8_t buf2[VW_MAX_MESSAGE_LEN];
    uint8_t buflen2;
    char ReplyBuffer[] = "ACK";
    int packetSize;
    mensaje Msg;
    buflen = VW_MAX_MESSAGE_LEN;
    buflen2 = VW_MAX_MESSAGE_LEN;
    // Leemos los paquetes que se hayan recibido por ethernet
        packetSize = _UDP.parsePacket();
        if (packetSize) {

                // Realizamos la asociación
                _UDP.read(buf,buflen);
                _ipAsociado = _UDP.remoteIP();
                _puertoAsociado = _UDP.remotePort();
                _estado = ASOCIADO;

                //contestamos a la asociación con ACK
                _UDP.beginPacket(_UDP.remoteIP(), _UDP.remotePort());
                _UDP.write(ReplyBuffer);
                _UDP.endPacket();
                delay(50);
        }

    if (Serial.available()) // Non-blocking
    {
        // Encendemos el led de mensaje recibido mientras se procesa
        digitalWrite(LED_RX_PIN, HIGH);
        // Leemos el mensaje
        buflen2 = Serial.readBytes(buf2, buflen2);
        Msg.setMsg(buf2, buflen2);
        // Procesamos el mensaje
        procesarPER(Msg);
        // Apagamos el led
        digitalWrite(LED_RX_PIN, LOW);
    }
    // Reflejamos el estado en los leds
    switch (_estado) {
        case ESPERA:
                // Encendemos el led del estado ESPERA y apgamos ASOCIADO
                digitalWrite(LED_ASOCIADO_PIN, LOW);
                digitalWrite(LED_ESPERA_PIN, HIGH);
            break;
            case ASOCIADO:
                // Encendemos el led de estado ASOCIADO y apagamos ESPERA
```

```
                    digitalWrite(LED_ASOCIADO_PIN, HIGH);
                digitalWrite(LED_ESPERA_PIN, LOW);
            break;
        }
    // Actualizamos el contador de segundos
        _contSegundos = _contSegundos + _incrementoTemporal;
        if(_contSegundos > 59){
                _contSegundos = _contSegundos - 60;
                if (ASOCIADO == _estado) {
                // En caso de estar asociado, enviamos ANUNCIAR cuando toque
                        _contAnuncio--;
                        if(0 == _contAnuncio){
                                _contAnuncio = T_ANUNCIO;
                                anunciar();
                        }
                }
        }
        delay(1000*_incrementoTemporal);
}
```

Código 3: Método "actualizarNP()"

Parche del nodo pasarela

Como puede verse en los Códigos 2 y 3, el nodo pasarela no hace uso de la biblioteca VirtualWire. En su lugar envía y recibe mensajes por el puerto serie. Esto se debe a que las bibliotecas EthernetUDP y VirtualWire son incompatibles en una misma placa Arduino UNO. Para solucionar este problema, se ha añadido un nuevo elemento que no interfiere en el funcionamiento de los protocolos.

Este elemento será una placa Arduino que reenvíe los mensajes recibidos a través del puerto series por radio (empleando VirtualWire) y viceversa. De esta forma se evita usar la biblioteca EhternetUDP y VirtualWire en la misma placa Arduino. Desde el punto de vista lógico, ambas placas formarán un único nodo de tipo pasarela. El código de esta placa está implementado en el fichero WJSN_parche.ino y se puede ver en el Código 4.

```
/*
 * Nombre: WJSN_parche.ino
 *
 * Autor: Pablo Linares Serrano
 *
 * Descripción: Código para solucionar la incompatibilidad de la biblioteca
 * Ethernet y VirtualWire. La idea es que la pasarela envía a través del
 * puerto serie lo que quiera transmitir por radio y otra placa diferente lo
 * hará. El puente es a nivel físico prácticamente y no influye en la
 * conformación de los paquetes.
 */

#include <VirtualWire.h>

#define TASA 2000
#define TX_PIN 4

byte buf[VW_MAX_MESSAGE_LEN];
byte buflen = VW_MAX_MESSAGE_LEN;
byte buf2[VW_MAX_MESSAGE_LEN];
byte buflen2 = VW_MAX_MESSAGE_LEN;
int tam;

void setup() {
   Serial.begin(9600);
   // configuramos VW
   vw_setup(TASA);  // Bits per sec
```

```
    vw_set_ptt_inverted(true); // Required for DR3100

    // inicializamos el receptor
    vw_set_rx_pin (5);
    vw_rx_start();       // Start the receiver PLL running

    // inicializamos el transmisor
    vw_set_tx_pin(4);
}

void loop() {
  buflen = VW_MAX_MESSAGE_LEN;
  buflen2 = VW_MAX_MESSAGE_LEN;
  if(Serial.available()){
    digitalWrite(13, true); // Flash a light to show transmitting
    tam = Serial.readBytes(buf, buflen);
    vw_send((uint8_t *)buf, tam);
    vw_wait_tx(); // Wait until the whole message is gone
    digitalWrite(13, false);
  }
  if (vw_get_message(buf2, &buflen2)) // Non-blocking
  {
      Serial.write(buf2, buflen2);
      delay(1500);
  }
  delay(100);
}
```

Código 4: fichero WJSN_parche.ino.

4. Código de los nodos sensores

Fichero WJSN_sensor.ino

El código con el que se programará los nodos sensores está contenido en el fichero WJSN_sensor.ino, que puede verse en el Código 5. Se puede ver que es similar al del nodo pasarela, con la diferencia de que se emplean los métodos correspondientes al nodo sensor. Además, se realizan las medidas que serán enviadas a las pasarelas descubiertas. Para ello se emplea el interfaz descrito anteriormente mediante las variables “bandera” y “tam” junto con la tabla “datos”. Los punteros de estas variables serán pasados al objeto nodoSensor al inicializarlo.

```
/*
 *  Nombre: WJSN_sensor.ino
 *  Autor: Pablo Linares Serrano
 *
 *  Descripción: Implementación de un nodo sensor WJSN empleando la
 *  biblioteca WJSN.h
 */
// inclusión de bibliotecas. La biblioteca VW está incluida en WJSN.h
#include <WJSN.h>
#include <dht11.h>

// Definimos las etiquetas de los canales
#define ID_TEM 1
#define ID_HUM 2

// Definición de parámetros
#define ID 3
#define INCREMENTO 1
#define DHT11PIN 3

// Declaración de los punteros para pasar las medias al PGD
uint8_t bandera;
uint8_t tam;
uint8_t datos[4];
```

```
uint8_t estado;
// Declaración del objeto nodo WJSN
nodoSensor sensor;
// Declaración del objeto sensor de humedad/temperatura
dht11;
void setup() {
  sensor.inicializarNS(ID, INCREMENTO, &bandera, &tam, datos);
  Serial.begin(9600);
}

void loop() {
  sensor.actualizarNS();
  if (0 == bandera){
    //medimos
    if(DHT11.read(DHT11PIN)){
      datos[0] = ID_TEM;
      datos[1] = (uint8_t) DHT11.temperature;
      datos[2] = ID_HUM;
      datos[3] = (uint8_t) DHT11.humidity;
      tam = 2;
      bandera = 1;
      Serial.print("temperatura: ");
      Serial.println(String(DHT11.temperature));
      Serial.print("humedad: ");
      Serial.println(String(DHT11.humidity));
    }
  }
}
```

Código 5: Fichero WJSN_sensor.ino.

Fichero WJSN.cpp

El código de los métodos "inicializarNS()" y "actualizarNS()" está implementado en el fichero WJSN.cpp y pueden verse en el Código 6 y el Código 7, respectivamente.

```
void nodoSensor::inicializarNS(uint8_t id, uint8_t incrementoTemporal,
uint8_t* bandera, uint8_t* tam, uint8_t* datos){
       // Inicializa un nodo sensor otorgando los parámetros necesarios a la
       // instancia en tiempo de ejecución.
       // inicialización de los pines de los led
       pinMode(LED_RX_PIN, OUTPUT);
       pinMode(LED_TX_PIN, OUTPUT);
       pinMode(LED_ASOCIADO_PIN, OUTPUT);
       pinMode(LED_ESPERA_PIN, OUTPUT);
       // asignación de valores a variables
       _incrementoTemporal = incrementoTemporal;
       _id = id;
       _estado = ESPERA;
       _tamTablaRD = 0;
       _tamTablaRA = 0;
       _contSegundos = 0;
       _contDescubrir = T_DESCUBRIR;
       _contAnuncio = T_ANUNCIO;
       _contMedicion = T_MEDIDA;
    // Interfaz con el programa que toma medidas
    _bandera = bandera;
    _tam = tam;
    _datos = datos;
    // Parámetros y funciones de VirtualWire
    vw_setup(2000);  // Bits per sec
       vw_set_ptt_inverted(true); // Required for DR3100
       // inicializamos el receptor
       vw_set_rx_pin (RX_PIN);
       vw_rx_start();       // Start the receiver PLL running
       // inicializamos el transmisor
       vw_set_tx_pin(TX_PIN);
}
```

Código 6: Método "inicializarNS()".

```
void nodoSensor::actualizarNS(){
      // Actualiza todos los contadores necesarios,
      // Comprueba si se han recibido mensajes y responde de ser necesario
      // Comprueba si es necesario enviar nuevos mensajes

      // Declaramos el buffer de lectura de mensajes
      uint8_t buf[VW_MAX_MESSAGE_LEN];
    uint8_t buflen = VW_MAX_MESSAGE_LEN;
    mensaje Msg;
    //uint8_t tam;
    uint8_t* medida;
    uint8_t tamMedida;

    // Leemos los mensajes que se hayan recibido
    if (vw_get_message(buf, &buflen)) // Non-blocking
    {
      // Encendemos el led de mensaje recibido mientras se procesa
      digitalWrite(LED_RX_PIN, HIGH);
      Msg.setMsg(buf, buflen);
      procesarPER(Msg);

      // Apagamos el led
      digitalWrite(LED_RX_PIN, LOW);
    }
    switch (_estado) {
      case ESPERA:
            // Encendemos el led del estado ESPERA y apagamos ASOCIADO
            digitalWrite(LED_ASOCIADO_PIN, LOW);
            digitalWrite(LED_ESPERA_PIN, HIGH);
        break;
        case ASOCIADO:
          // Encendemos el led de estado ASOCIADO y apagamos ESPERA
                digitalWrite(LED_ASOCIADO_PIN, HIGH);
          digitalWrite(LED_ESPERA_PIN, LOW);
        break;
      }
    // Actualizamos el contador de segundos
      _contSegundos = _contSegundos + _incrementoTemporal;
      if(_contSegundos > 59){
            // Si se cumple un minuto, ajustamos el valor del contador y
            // disminuimos los demás contadores (que cuentan minutos)
            _contSegundos = _contSegundos - 60;
            switch (_estado){
            case ESPERA:
                        _contDescubrir--;

                        if(0 == _contDescubrir){
                        // Si el contador llega a cero lo reseteamos
                        // y enviamos un mensaje descubrir
                              _contDescubrir = T_DESCUBRIR;
                              descubrir();
                        }
                  break;

                  case ASOCIADO:
                      _contAnuncio--;
                        _contMedicion--;

                      if(0 ==_contMedicion){
                        // Comprueba que el usuario
                        // haya indicado que hay una
                        // nueva medida disponible y la
                        // envía en caso afirmativo.
                          _contMedicion = T_MEDIDA;
                          medida = medir(&tamMedida);
                          Msg.setMsg(medida, tamMedida);
                          envioA(Msg);
                          free(medida);
```

```
                        }
                        if(0 == _contAnuncio){
                           //reseteamos el contador y
                           // enviamos un anuncio
                           _contAnuncio = T_ANUNCIO;
                           anunciar();
                        }
                    break;

                default:
                break;
                }
        }
        delay(1000*_incrementoTemporal);
}
```

Código 7: Método "actualizarNS()".

5. Construcción de los nodos y red desplegada

Nodo pasarela

El nodo pasarela no estará equipado con ningún sensor. Como se ha indicado anteriormente, estará formado por dos tarjetas Arduino por las incompatibilidades mencionadas entre las bibliotecas VirtualWire y Ethernet. Ambas tarjetas se comunicarán mediante sus puertos serie. Una gestionará solamente la comunicación por radio, haciendo de repetidor entre el puerto serie y el interfaz radio y viceversa. Esta tarjeta será una Arduino Nano. El resto de funciones del nodo tendrán lugar en la tarjeta Arduino Uno, que ejecutará todos los protocolos y gestionará el interfaz Ethernet. Se puede ver un esquemático del nodo pasarela en la Figura 1 y una fotografía del prototipo construido en la Figura 2.

Nodo Sensor

Los nodos sensores tendrán un periférico para transmitir mensajes por radio y otro para recibirlos. Además, constarán de un sensor de humedad y temperatura h11. En la Figura 3 se puede ver el esquemático de los nodos sensores y en la Figura 4 una imagen del prototipo que se ha construido.

Red desplegada

Para comprobar el funcionamiento de los protocolos y los nodos programados, se han implementado dos nodos sensores y un nodo pasarela como los descritos. Además, se ha programado un cliente en Python (ver Código 8) que se conectará con la pasarela para recibir las mediciones de los nodos. La topología lógica de la red dependerá de las circunstancias en las que se inicialice y de las pérdidas de paquetes que tengan lugar. Sin embargo, lo más probable es que ambos nodos se asocien con la pasarela directamente.

Para visualizar los intercambios de paquetes entre nodos, se ha programado un sniffer (ver Código 9). Éste código se le puede subir a uno de los nodos sensores. Al hacerlo, escuchará el tráfico de los demás nodos y enviará los paquetes interceptados a través del puerto serie. Sin embargo, dejará de comportarse como un nodo sensor.

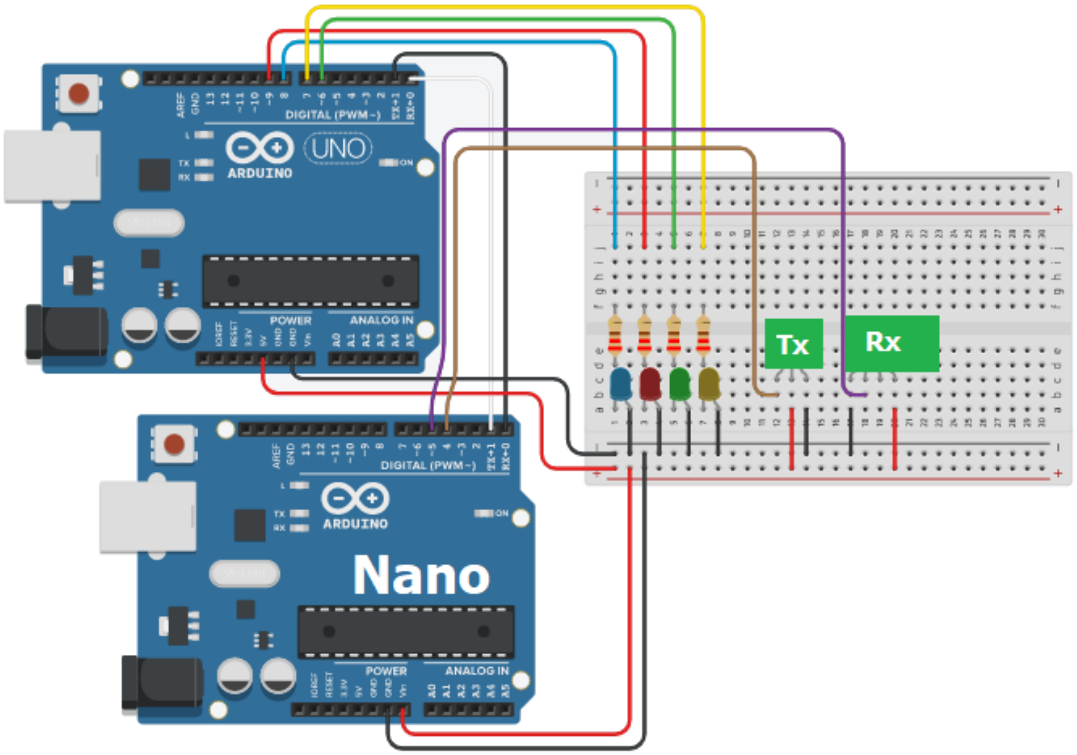

Figura 1: Esquemático de un nodo pasarela.

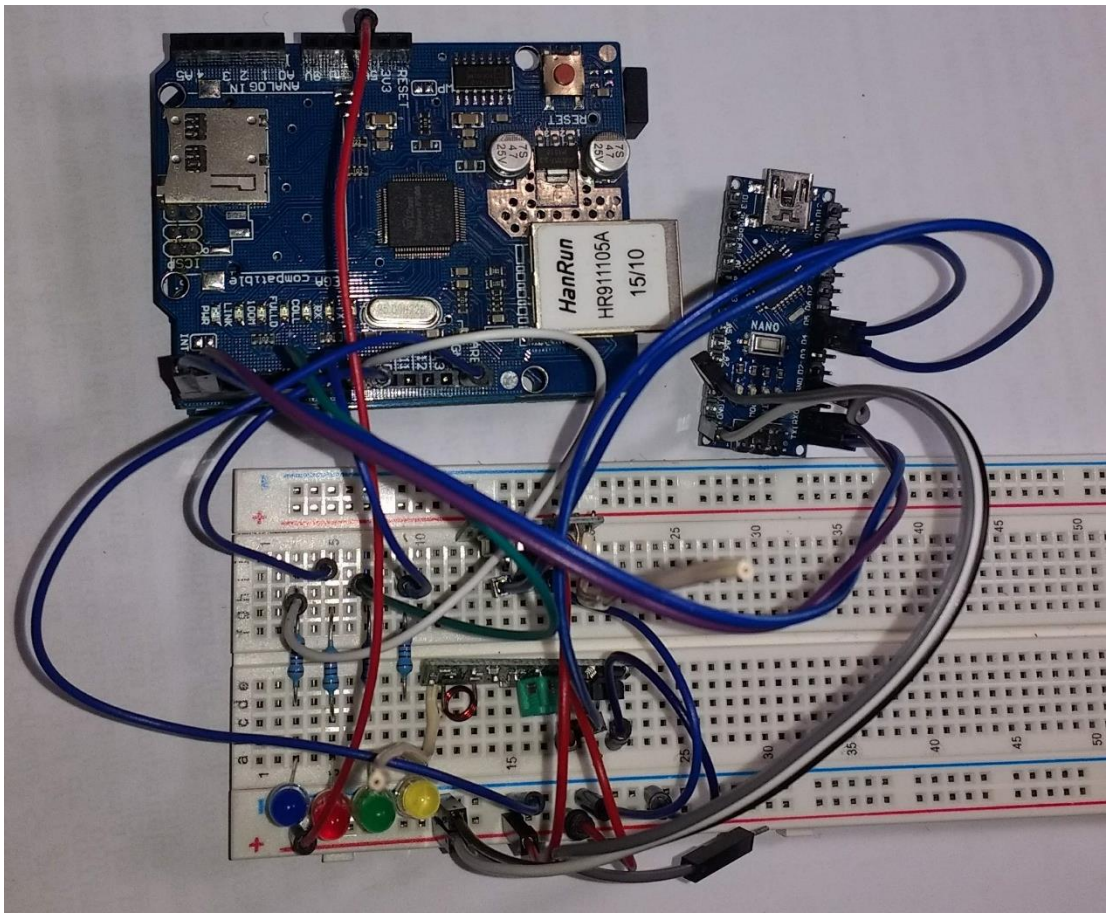

Figura 2: Fotografía del prototipo de nodo pasarela.

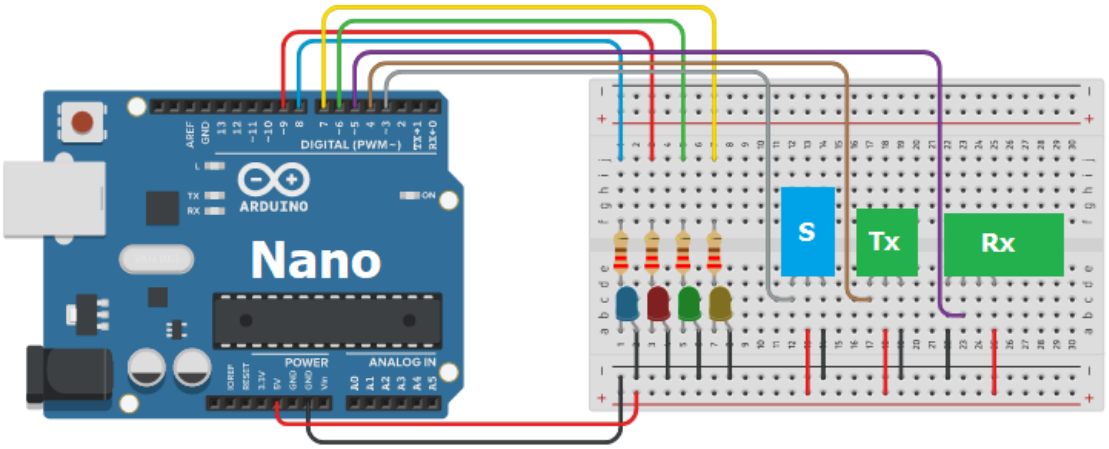

Figura 3: Esquemático de un nodo sensor.

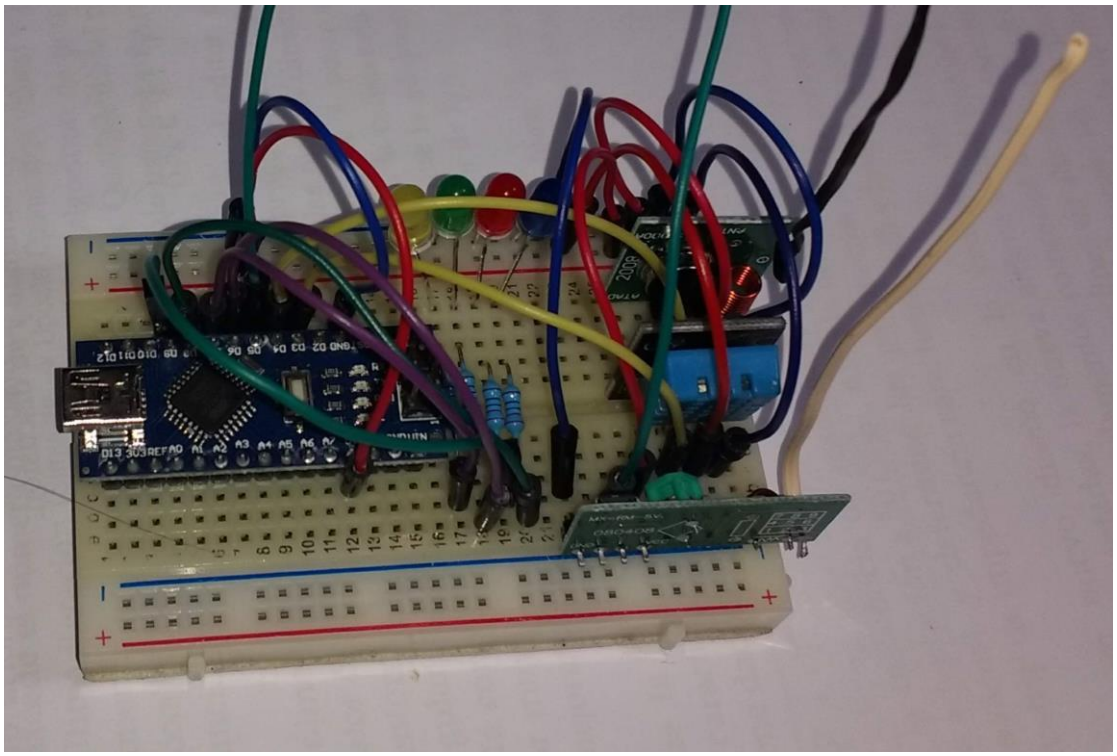

Figura 4: Fotografía de uno de los nodos sensores construidos.

```
#
#      Autor: Pablo Linares
#
#      Descripción: Ejemplo de cliente sencillo para la red WJSN.
Escrito en Python 3
#
#      Fecha: 15/1/2021
#

import time
import socket

HostIP = "169.254.1.177"
```

```
Npuerto = 5027
Mensaje = b"ASOCIAR"

puerto = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#puerto.bind(("169.254.141.235", 53048))
puerto.sendto(Mensaje, (HostIP, Npuerto))

while True:
      fichero = open("recibido.txt", "a+")
      data, addr = puerto.recvfrom(1024)
      fichero.write("Hora: ")
      fichero.write(time.ctime())
      fichero.write("; Recibido: ")
      fichero.write(str(list(data)))
      fichero.write("; origen: ")
      fichero.write(str(addr))
      fichero.write(";\n")
      fichero.close()
```

Código 8: Ejemplo de cliente sencillo programado en Python.

```
*
 *  Nombre: WJSN_sensor_debug.ino
 *  Autor: Pablo Linares Serrano
 *
 *  Descripción: Código para programar un nodo sensor como debugger de otro
nodo sensor.
 *  Permite enviarle mensajes y leer los que envíe.
 */

// inclusión de bibliotecas. La biblioteca VW está incluida en WJSN.h
#include <WJSN.h>
#include <VirtualWire.h>
#define TX_PIN 4
#define RX_PIN 5
#define LED1 13
#define LED2 8
#define ID_LOC 1
#define ID_REM 2
#define DIF 255
#define DD 254

// Definimos los tipos de mensajes CEPER
#define ANUNCIAR 1
#define CONFIGURAR 2
#define DESCUBRIR 3
#define ASOCIAR 4
#define ACK 5
#define RECHAZAR 6

char rx;
uint8_t tam;
uint8_t buf[200];
uint8_t buflen = 200;
mensaje Msg(buf, buflen);

void setup() {
  Serial.begin(9600);
  Serial.println("Setup");
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  rx = '3';
  tam = 0;
  vw_setup(2000);  // Bits per sec
  vw_set_ptt_inverted(true); // Required for DR3100
```

```
  // inicializamos el receptor
  vw_set_rx_pin (RX_PIN);
  vw_rx_start();       // Start the receiver PLL running
  // inicializamos el transmisor
  vw_set_tx_pin(TX_PIN);
}

void loop() {
  buflen = 200;
  if (vw_get_message(buf, &buflen)) // Non-blocking
  {
    int i;
    digitalWrite(LED1, HIGH);
    Serial.println("Got message: ");
    for (i = 0; i < buflen; i++)
    {
      Serial.print(String(buf[i]));
      Serial.print(" ");
    }
    Serial.println("");
    Serial.println("Got 2: ");
    Serial.write(buf, buflen);
    Serial.println("");
    Msg.setMsg(buf, buflen);
    Serial.println("SIGS: " + String(Msg.getSigs()));
    Serial.println("DEST: " + String(Msg.getDest()));
    Serial.println("ORIG: " + String(Msg.getOrig()));
    if (Msg.getProt()== 1){
      Serial.println("Protocolo encapsulado: CEPER");
      switch (Msg.getTipo()){
        case ANUNCIAR:
          Serial.println("Tipo: ANUNCIAR");
          break;
        case CONFIGURAR:
          Serial.println("Tipo: CONFIGURAR");
          break;
        case DESCUBRIR:
          Serial.println("Tipo: DESCUBRIR");
          break;
        case ASOCIAR:
          Serial.println("Tipo: ASOCIAR");
          break;
        case ACK:
          Serial.println("Tipo: ACK");
          break;
        case RECHAZAR:
          Serial.println("Tipo: RECHAZAR");
          break;
      }
    }
    Serial.println("_____________________________________________");
    digitalWrite(LED1, LOW);
  }
  delay(50);
}
```

Código 9: Fichero WJSN_sniffer.ino.