IoT Lab 3

Long distance communication with Remote Terminals over LoRa radio links

3.1 Essential Features of LoRa Communication Technology

LoRa (Long Range) is a wireless communication technology designed for long-range, low-power, and low-data-rate applications, often used in Internet of Things (IoT) networks.

It is based on **Chirp Spread Spectrum** (CSS) modulation and offers several unique features that make it ideal for specific use cases.

1. **Long Range Communication**
   - LoRa supports communication over long distances, ranging from **2 km in urban areas** to **15–20 km in rural areas**, depending on environmental conditions and antenna setup.
   - This long-range capability makes it suitable for applications like agriculture, smart cities, and remote monitoring.

2. **Low Power Consumption**
   - LoRa devices are optimized for low-power operation, enabling **battery-powered sensors to last for years** (often 10+ years) on a single charge.
   - This makes it ideal for applications where power supply is limited or replacement is challenging.

3. **Wide Area Networking**
   - LoRa operates on unlicensed ISM bands (e.g., 433 MHz, **868 MHz**, 915 MHz), which allows it to cover wide areas without requiring costly licensed spectrum.
   - LoRaWAN (a higher-layer protocol) is often used for network management in such setups.

4. **Robustness to Interference**
   - LoRa uses chirp spread spectrum modulation, which spreads data over a wide bandwidth, making it highly resistant to noise and interference.
   - It can maintain communication even in environments with high RF noise.

5. **Scalability**
   - LoRa networks can support thousands of devices, making it well-suited for IoT applications where multiple sensors and actuators need to operate within a single network.

6. **Low Data Rate**
   - LoRa is designed for low-data-rate applications, with typical throughput ranging from **0.3 kbps to 50 kbps**.
   - Suitable for transmitting small packets of data, such as sensor readings, but not for high-bandwidth tasks like video streaming.

7. **Bidirectional Communication**
   - Supports both uplink (sensor-to-gateway) and downlink (gateway-to-sensor) communication.
   - Allows devices to receive commands, firmware updates, or acknowledgments.

8. **Multi-Gateway Support**
   - LoRa devices can communicate with multiple gateways simultaneously, ensuring reliable data transmission even in challenging environments.

9. **Secure Communication**
    - Offers built-in security features, such as **AES-128 encryption**, to protect data from interception or tampering.
    - Essential for applications in industries like healthcare and finance.

10. **License-Free Operation**
    - Operates in license-free sub-GHz ISM bands (e.g., 433 MHz, 868 MHz, 915 MHz), eliminating the need for costly spectrum licensing.
    -

11. **Flexible Deployment Models**
    - LoRa can be used in both private and public networks.
    - Public networks are ideal for smart cities, while private networks are better for industries and agriculture.

Key Use Cases for LoRa

1. **Smart Agriculture**:
    - Soil moisture sensors, livestock tracking, weather stations.
2. **Smart Cities**:
    - Street lighting control, parking sensors, air quality monitoring.
3. **Industrial IoT**:
    - Predictive maintenance, asset tracking, energy management.
4. **Environmental Monitoring**:
    - River levels, forest fire detection, wildlife monitoring.
5. **Healthcare**:
    - Remote patient monitoring and wearable devices.

Comparison with Other Technologies

Feature	LoRa	Wi-Fi	Bluetooth	Cellular (LTE/5G)
Range	Long (up to 20 km)	Short (< 100 m)	Short (< 10 m)	Moderate to Long
Power Consumption	**Very Low**	High	Low	**High**
Data Rate	Low	High	Moderate	High
Deployment Cost	Low	Moderate	Low	High
Network Size	**High**	Low	Low	High

Advantages of LoRa

1. **Extensive Coverage**: Excellent for remote areas and large-scale deployments.
2. **Energy Efficiency**: Ideal for battery-powered IoT devices.
3. **Cost-Effective**: Uses free ISM bands and requires minimal infrastructure.
4. **Scalability**: Suitable for networks with thousands of devices.

Limitations of LoRa

1. **Low Throughput**: Not suitable for high-data-rate applications.
2. **Latency**: Not designed for real-time communication.
3. **Interference in Dense Networks**: Performance can degrade with too many devices in the same area.

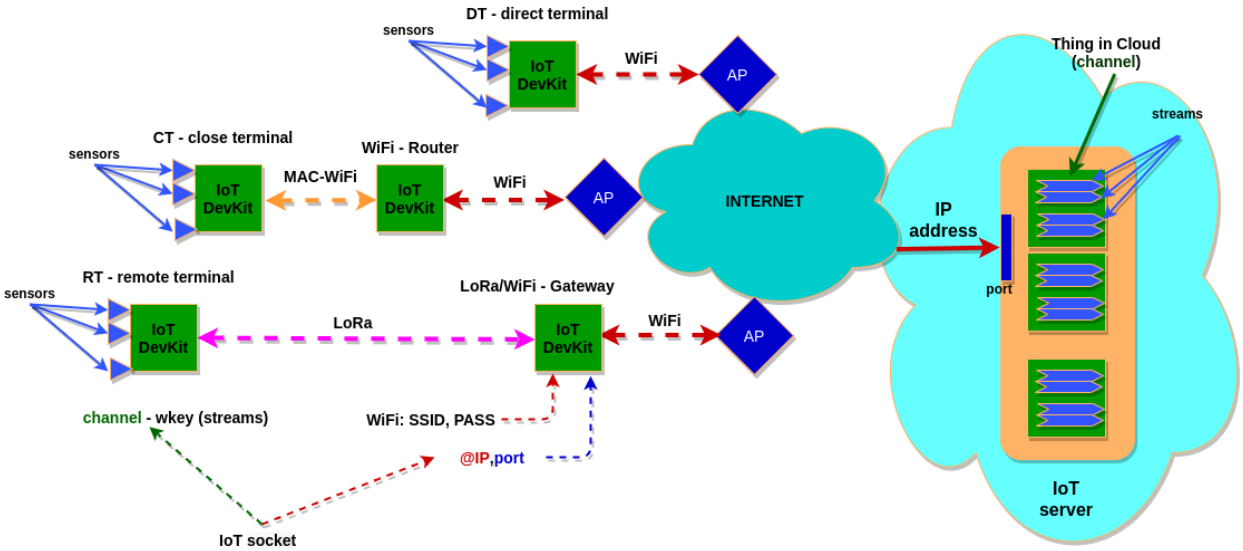

Fig 3.1 Sending sensor data from **Remote Terminal** via LoRa-WiFi Gateway to ThingSpeak server. The IoT socket parameters decomposition: **channel** in Remote **Terminal**, **@IP and port number** in **LoRa-WiFi Gateway**.

3.1 Simple LoRa sender and receiver nodes

The following examples present the introduction to developing LoRa based communication on our DevKits that integrate LoRa -**SX127x** modems.

Below we have a very simple example including one sender and one receiver that must operate on the same frequency – **`freq`** (ex. **868MHz**), with the same spreading factor -**`sf`** (ex. **7**) , the same bandwidth – **`bw`** (ex. **125KHz**), and the same coding ratio – **`cr`** (ex. **5**).

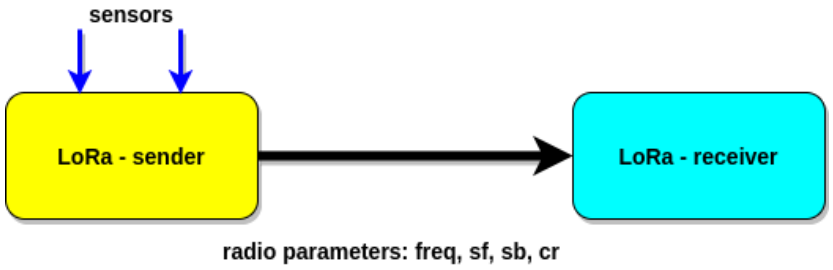

Fig. 3.2 Simple IoT architecture with sender-receiver nodes and LoRa radio link

To prepare the LoRa modems we have to provide initial configuration code required to connect and activate the modems.

3.1.1 LoRa modem initialization parameters and function

The following code contains configuration functions. It is called **`lora_init.py`**. When we run this code we "connect" ,via SPI bus, our **ESP32C3** SoC to the external modem and we send the initialization commands to the modem.

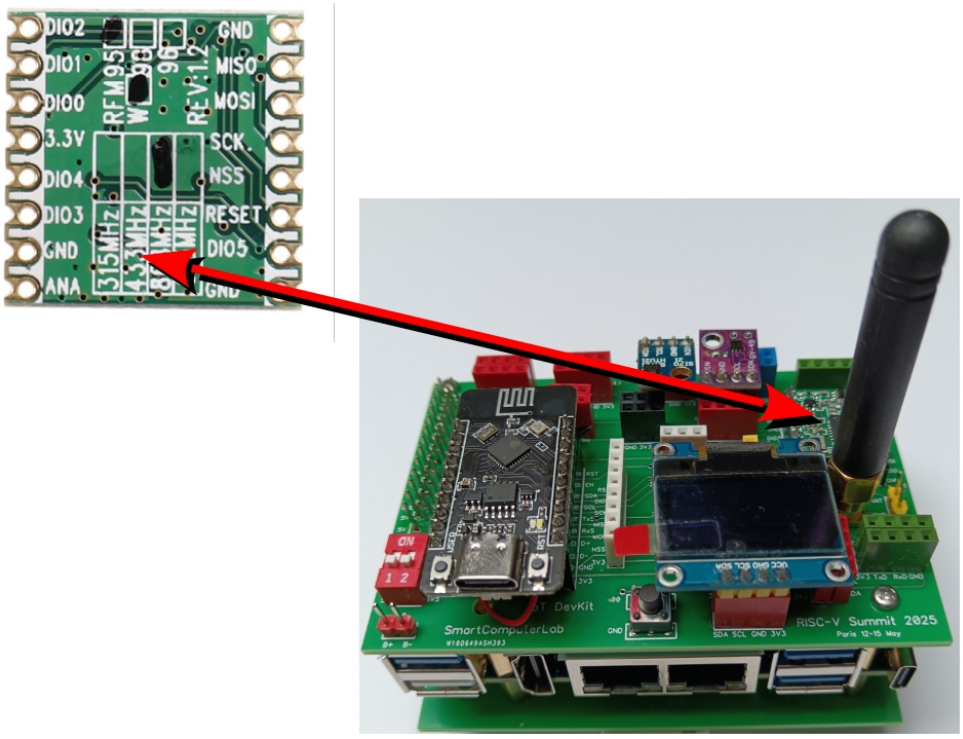

Fig 3.3 IoT DevKit with HT ESP32C3 board and LoRa **sx1276 modem** on **RFM95 module** with SPI bus (**`sck`**,**`miso`**,**`mosi`**) and control lines: **DIO0** (**`dio_0`**), **NSS** (**`ss`**), **RESET** (**`reset`**).

```
-----------------------------------------------------------------------------------------------------------
#lora_init.py
from machine import Pin, SPI
import time
import sx127x  # SX127x LoRa driver (ensure the library is installed)
import esp32

# --- LoRa Pins and SPI Bus Setup --- HT
lora_pins = {
    'dio_0': 2,         # DIO0 pin for interrupt
    'ss': 4,            # Slave Select (SS)
    'reset': 10,        # Reset pin
    'sck': 6,           # SPI Clock pin
    'miso': 5,          # SPI MISO pin
    'mosi': 7           # SPI MOSI pin
}
# SPI bus configuration for SX1276
lora_spi = SPI(
    baudrate=10000000,     # Set baudrate to 10 MHz
    polarity=0,            # Clock polarity (CPOL)
    phase=0,               # Clock phase (CPHA)
    bits=8,                # 8 bits per transfer
    firstbit=SPI.MSB,      # MSB first
    sck=Pin(lora_pins['sck'], Pin.OUT, Pin.PULL_DOWN),    # SCK (clock)
    mosi=Pin(lora_pins['mosi'], Pin.OUT, Pin.PULL_UP),    # MOSI (Master Out Slave In)
    miso=Pin(lora_pins['miso'], Pin.IN, Pin.PULL_UP),     # MISO (Master In Slave Out)
)
# LoRa configuration with default parameters
lora_default = {
    'frequency': 868E6,             # Frequency for Europe (868 MHz ISM band)
    'tx_power_level': 14,           # Transmission power level (14 dBm)
    'signal_bandwidth': 125E3,      # Signal bandwidth (125 kHz)
    'spreading_factor': 7,          # Spreading factor (7)
    'coding_rate': 5,               # Coding rate (4/5)
    'preamble_length': 8,           # Preamble length (8)
    'implicit_header': False,       # Explicit header mode
    'sync_word': 0x12,              # LoRa sync word
    'enable_crc': True              # Enable CRC for error detection
}
```

```
# --- SX1276 LoRa Driver Initialization ---
def lora_init():
    # Reset the SX1276
    reset_pin = Pin(lora_pins['reset'], Pin.OUT)
    reset_pin.value(0)
    time.sleep(0.01)  # Short delay
    reset_pin.value(1)
    # Initialize the SX1276 LoRa driver with default parameters
    lora = sx127x.SX127x(spi=lora_spi, pins=lora_pins, parameters=lora_default)
    # Confirm initialization
    print("LoRa modem initialized with default parameters.")
    return lora

#lora_init()                        # only for test
```

```
MPY: soft reboot
SX version: 18
LoRa modem initialized with default parameters.
```

To do:

Attention: Correct initialization of the modem/module prints: `SX version: 18`

Analyze the above code , distinguish 3 parts: connection, default radio link parameters (may be modified), and the initial initialization run with `lora_init()` function.

3.1.2 LoRa sender - `main_send_lora.py` and recv ACK

In the following example we send data packets over LoRa link. The format of the packets is the same as in the already presented ESP-NOW sender nodes (see Lab 4). The data packet contains 32 bytes.

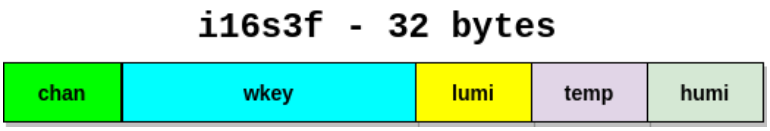

Fig. 3.4 Structure of LoRa packets carrying the channel number, write key/topic, and sensor values.

Note that the code below integrates **both sending and receiving functions**. The receiving function is used to capture the **return acknowledgment (ACK) packet**. This return packet does **not** carry sensor data; instead, it confirms that the data packet has been successfully received and may also include **control information**, which will be examined in the following labs.

Reception of the return packet is handled through a **callback function** that operates independently of the main operational loop implemented in the `main()` function. This callback is triggered by an **interrupt** signaling the arrival of a new **LoRa packet** in the modem's input buffer.
This interrupt-driven reception mechanism ensures timely handling of incoming packets without blocking the main application flow.

```
-----------------------------------------------------------------------------------------------------
import time, ustruct
from machine import I2C, Pin, deepsleep
from sensors import sensors
from lora_init import lora_init
# Initialize LoRa communication
lora = lora_init()

def onReceive(lora_modem,payload):
    #print("Waiting for LoRa packets...")
    ACK=payload.decode()
    print("Received LoRa packet:"+str(ACK))   #, payload.decode())
# Function to send sensor data over LoRa
def send_lora_data(l, t, h):
    try:
        # Create the message with temperature, humidity, and luminosity
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        # prepare data packet with bytes
        data = ustruct.pack('i16s3f', 1254,'smartcomputerlab',l,t,h)
        # Convert message to bytes
        # lora.println(bytes(message, 'utf-8'))
        lora.println(data)
        print("LoRa packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)
# Main program
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
def main():
    lora.onReceive(onReceive)
    lora.receive()
    while True:
        # Capture sensor data
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C"); print("Humidity:", humi, "%")
        # Send sensor data over LoRa
        send_lora_data(lumi, temp, humi)
        lora.receive()
        time.sleep(ACK_wait_time)          # waiting for ACK frame
        #lora.sleep()                     # only for deepsleep
        #deepsleep(10*1000)               # 10*1000 miliseconds
# Run the main program
main()
-----------------------------------------------------------------------------------------------------
```

3.1.3 LoRa receiver – and sender ACK

Below we have corresponding receiver node that displays the received data on its OLED screen. It responds with a simple ACK packet (message) to confirm the reception of the data packet.

```
from machine import Pin, I2C, SPI
import ustruct, time
from lora_init import *
from sensors_display import *
lora_modem = lora_init()
# --- Receive LoRa Packet ---
def onReceive(lora_modem,payload):
    rssi = lora_modem.packetRssi()
    chan, wkey, lumi, temp, humi = ustruct.unpack('i16s3f', payload)
    print("Received LoRa packet RSSI:"+str(rssi)); print(chan,wkey,lumi,temp,humi)
    sensors_display(8,9,lumi,temp,humi,0)
    lora_modem.println("ACK_packet")  # sending ACK packet
    lora_modem.receive()

def main():
    lora_modem.onReceive(onReceive)
    lora_modem.receive()
    while True:
        time.sleep(2)
        print("in the loop")

main()
```

```
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
import time

def sensors_display(sda, scl, luminosity, temperature, humidity, duration):
    i2c = I2C(scl=Pin(scl), sda=Pin(sda), freq=400000)
    oled = SSD1306_I2C(128, 64, i2c)
    oled.fill(0)
    oled.text("Sensor readings", 0, 0)
    oled.text("Lux: {:.2f}".format(luminosity), 0, 16)
    oled.text("Temp: {:.2f}".format(temperature), 0, 32)
    oled.text("Humi: {:.2f}".format(humidity), 0, 48)
    oled.show()
    if duration!=0:
        time.sleep(duration)
        oled.poweroff()
```

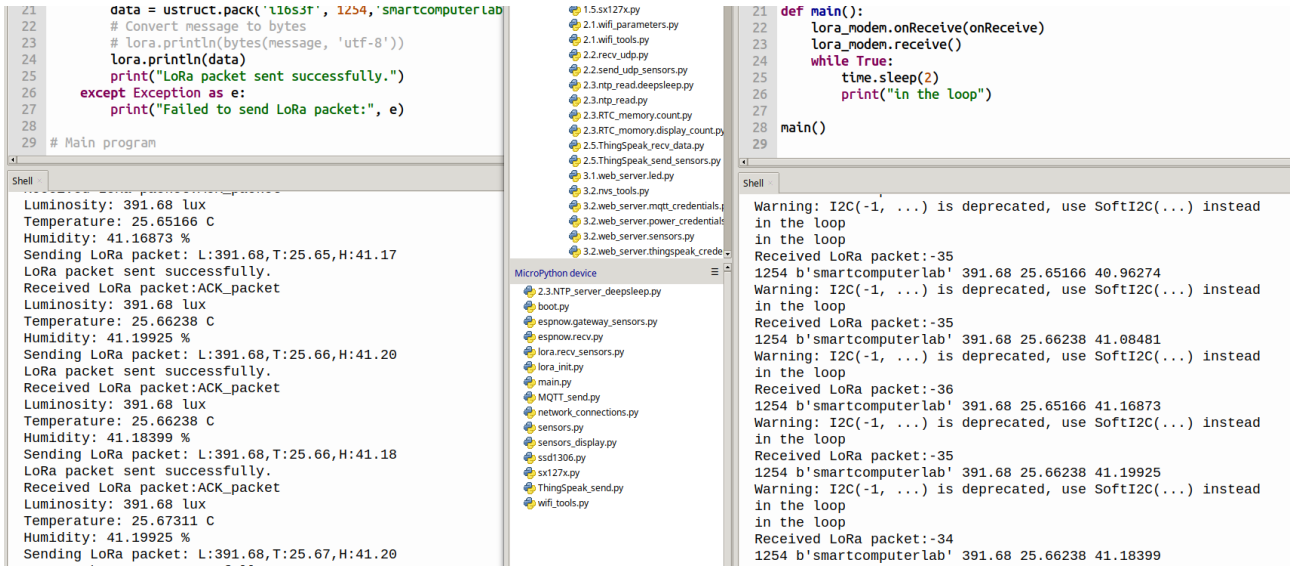

Fig. 3.5 LoRa sender and receiver nodes; note the use of ACK packet

To do

Run the Remote Terminal with deepsleep mode for `low_power` stage

3.2 Sending/receiving formatted data/ACK packets with AES encryption

The following example is a **minor modification** of the previous one; however, it introduces the use of **power-control parameters** delivered through the **ACK packet**.
The formatted ACK packet contains **four fields**:

1. The **channel number** (topic), which also serves as the **terminal identifier**,
2. The **`cycle` duration** (in seconds),
3. The **`delta` value**, indicating the minimum difference between the last transmitted sensor reading and the current one (e.g., temperature),
4. The **`kpack` value**, which specifies the maximum number of consecutive measurement cycles allowed without transmitting a data packet.

All of these parameters are used to **adaptively reduce the average current consumption** of the remote terminal by controlling its transmission frequency and sensitivity.
These formatted ACK packets are **encrypted using the AES encryption scheme**, ensuring secure delivery of control information.

AES

AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used for secure data transmission. It was established by the U.S. National Institute of Standards and Technology (NIST) in 2001 and is recognized for its efficiency and high level of security. AES uses the same key for both encryption and decryption, making it a **symmetric encryption** method.

Key features of AES include:

1. **Block Cipher**: It encrypts data in **fixed-size blocks** (**128 bits**).
2. **Key Sizes**: Supports key sizes of 128, 192, or 256 bits, providing varying levels of security.
3. **Efficient and Fast**: Designed to perform efficiently in both hardware and software environments.
4. **Widely Adopted**: Trusted by industries, governments, and organizations for secure communications.

3.2.1 Sending data packets and receiving ACK packets (AES encrypted)

We start this code with the introduction of AES functions; they are prepared in **`aes_tools.py`** module as follows:

3.2.1.1 AES module – `aes_tools.py`

```
-----------------------------------------------------------------------------------------------------
# aes_tools.py
import ucryptolib

# AES encryption function using ucryptolib
def aes_encrypt(data,aes_key):
    cipher = ucryptolib.aes(aes_key, 1)  # 1 = ECB mode
    encrypted = cipher.encrypt(data)     # data size  must be multiple 16 bytes
    return encrypted
# AES decryption function
def aes_decrypt(encrypted_data,aes_key):
    cipher = ucryptolib.aes(aes_key, 1)  # 1 = ECB mode
    data = cipher.decrypt(encrypted_data)   # data size  must be multiple 16 bytes
    return data
-----------------------------------------------------------------------------------------------------
```

3.2.1.2 The sender module (with AES) code

```
-----------------------------------------------------------------------------------------------------
import time, ustruct
from machine import I2C, Pin, deepsleep
from sensors import sensors
from lora_init import lora_init
from aes_tools import *
AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
# Initialize LoRa communication
lora = lora_init()
chan = 1234

def onReceive(lora_modem,payload):
    if len(payload)==16:
        ack=aes_decrypt(payload,AES_KEY)
        rchan, cycle, delta, kpack = ustruct.unpack('2ifi', ack)
```

```
        print("encrypted ACK received");
        if chan==rchan :
            print(cycle,delta,kpack)

# Function to send sensor data over LoRa
def send_lora_data(l, t, h):
    try:
        # Create the message with temperature, humidity, and luminosity
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        # prepare data packet with bytes
        data = ustruct.pack('i16s3f', chan,'smartcomputerlab',l,t,h)
        enc_data=aes_encrypt(data,AES_KEY)
        # Convert message to bytes
        # lora.println(bytes(message, 'utf-8'))
        lora.println(enc_data)
        print("LoRa encrypted packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)

# Main program
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
def main():
    lora.onReceive(onReceive)
    lora.receive()
    while True:
        # Capture sensor data
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        # Send sensor data over LoRa
        send_lora_data(lumi, temp, humi)
        lora.receive()
        time.sleep(ACK_wait_time)           # waiting for ACK frame
        #lora.sleep()                      # only for deepsleep
        #deepsleep(10*1000)                # 10*1000 miliseconds

# Run the main program
main()
```

3.2.1.3 Receiving data packets and sending ACK packets (AES encrypted)

```
from machine import Pin, I2C, SPI
import ustruct
from lora_init import *
from sensors_display import *
from aes_tools import *
import time

AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
# Initialize LoRa modem
lora_modem = lora_init()

# --- Receive LoRa Packet ---
def onReceive(lora_modem,payload):
    rssi = lora_modem.packetRssi()
    if len(payload)==32:
        rssi = lora_modem.packetRssi()
        data=aes_decrypt(payload,AES_KEY)
        chan, wkey, temp, humi, lumi = ustruct.unpack('i16s3f', data)
        print("Received encrypted LoRa packet with RSSI: "+str(rssi))   #, payload.decode())
        print(chan,wkey,lumi,temp,humi)
        sensors_display(8,9,lumi,temp,humi,0)
        ack=ustruct.pack('2ifi',chan,10,0.01,10)  # chan, cycle, delta, kpack
        enc_ack=aes_encrypt(ack,AES_KEY)
        lora_modem.println(enc_ack)  # sending ACK packet
        print("send encrypted ack AES packet")
        lora_modem.receive()

def main():
    lora_modem.onReceive(onReceive)
    lora_modem.receive()
    while True:
        time.sleep(2)
        print("in the loop")

main()
```

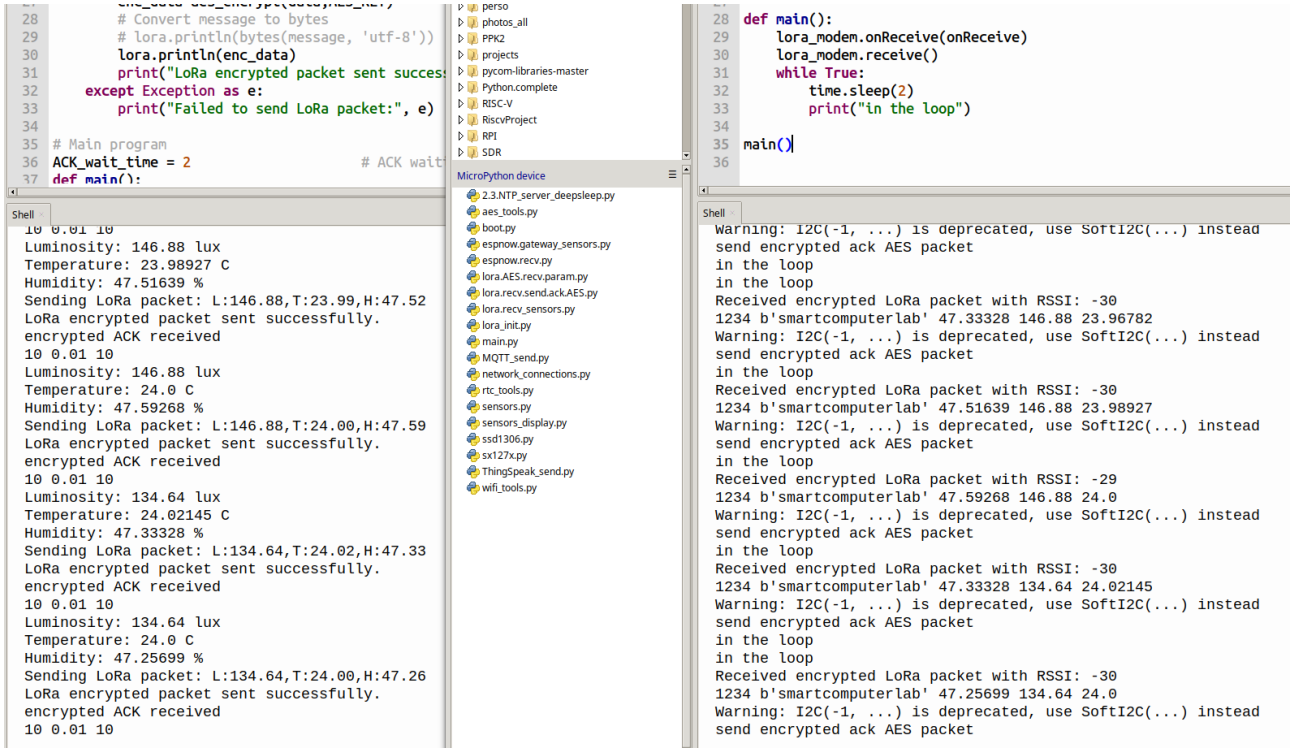

Fig. 3.6 LoRa sender and receiver nodes with the use of ACK packets and AES encryption

3.3 Building LoRa-WiFi gateways

Now we have operational link with data and ACK packets so ,at the receiver side, we can add the relay transmission via WiFi connection to MQTT broker or to ThingSpeak server.

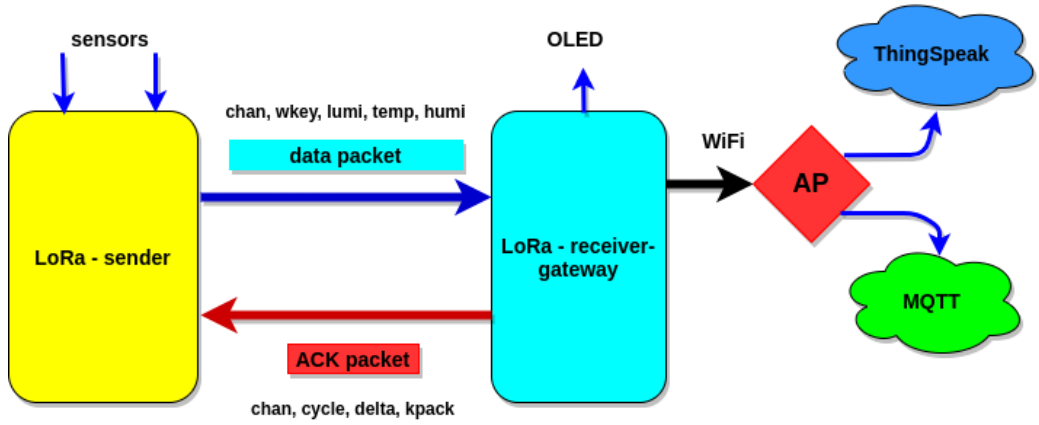

Fig 3.7 IoT architecture with Remote Terminal and LoRa-WiFi gateway to MQTT broker or/and ThingSpeak server.

3.3.1 LoRa-WiFi gateway to MQTT broker

Let us start with terminal node that sends an encrypted data packet every n seconds and first waits for the corresponding encrypted ack packet during **`ACK_wait_time`**. Then after few seconds the node sends next data packet.

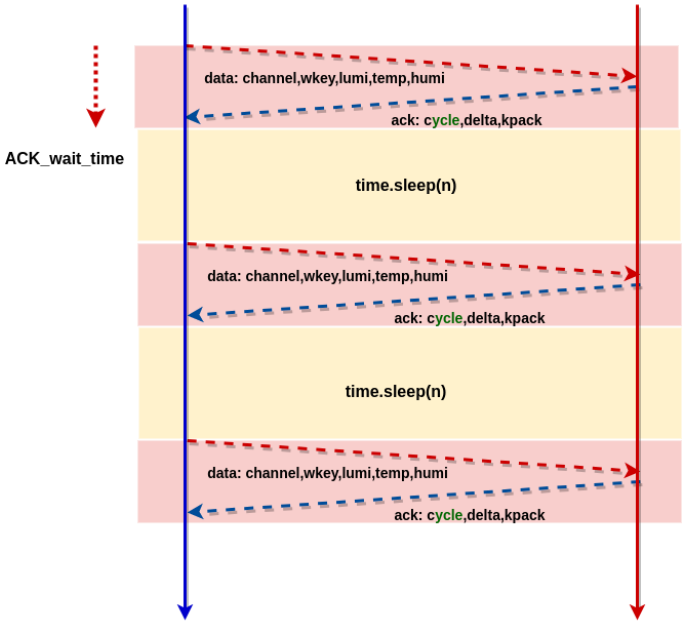

Fig 3.8 Simple sending (**`data`**) – receiving (**`ack`**) cycle with **`high_power`** wait time

---

3.3.1.1 LoRa sender node

```
import time, ustruct
from machine import I2C, Pin, deepsleep
from sensors import sensors
from lora_init import lora_init
from aes_tools import *
AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
# Initialize LoRa communication
lora = lora_init()
chan = 1234
led=Pin(3)
cycle=10

def onReceive(lora_modem,payload):
    global cycle
    if len(payload)==16:
```

```
        ack=aes_decrypt(payload,AES_KEY)
        rchan, cycle, delta, kpack = ustruct.unpack('2ifi', ack)
        print("encrypted ACK received");
        if chan==rchan :
            print(cycle,delta,kpack)

# Function to send sensor data over LoRa
def send_lora_data(l, t, h):
    try:
        # Create the message with temperature, humidity, and luminosity
        message = f"L:{l:.2f},T:{t:.2f},H:{h:.2f}"
        print("Sending LoRa packet:", message)
        # prepare data packet with bytes
        data = ustruct.pack('i16s3f', chan,'smartcomputerlab',l,t,h)
        enc_data=aes_encrypt(data,AES_KEY)
        lora.println(enc_data)
        print("LoRa encrypted packet sent successfully.")
    except Exception as e:
        print("Failed to send LoRa packet:", e)

# Main program
ACK_wait_time = 2                    # ACK waiting time depends on the protocol and data rate
def main():
    lora.onReceive(onReceive)
    lora.receive()
    led.off()
    while True:
        led.on()
        lumi, temp, humi = sensors(sda=8, scl=9)
        print("Luminosity:", lumi, "lux")
        print("Temperature:", temp, "C")
        print("Humidity:", humi, "%")
        # Send sensor data over LoRa
        send_lora_data(lumi, temp, humi)
        lora.receive()
        time.sleep(ACK_wait_time)           # waiting for ACK frame
        led.off()
        print(cycle)
        if cycle<600 :                       # to high value
            time.sleep(cycle)
        else:
            time.sleep(15)
        #lora.sleep()                       # only for deepsleep
        #deepsleep(10*1000)                 # 10*1000 miliseconds

# Run the main program
main()
```

3.3.1.2 LoRa-WiFi to MQTT gateway node

```
from machine import Pin, I2C, SPI
import ustruct, random, ubinascii
from lora_init import *
from sensors_display import *
from aes_tools import *
import machine,time
from wifi_tools import *
from umqtt.simple import MQTTClient
SSID = 'Bbox-9ECEBF79'
PASS = '54347A3EA6A1D6C36EF6A9E5156F7D'
# MQTT broker details
MQTT_BROKER = "broker.emqx.io"  # Replace with your broker address
MQTT_PORT = 1883
MQTT_CLIENT_ID = ubinascii.hexlify(machine.unique_id())  # Unique client ID
MQTT_TOPIC = 'esp32/sensor_data'  # Replace with your topic
# Initialize MQTT client
client = MQTTClient(MQTT_CLIENT_ID, MQTT_BROKER, port=MQTT_PORT)
AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
lora_modem = lora_init()

def connect_mqtt():
    """Connect to the MQTT broker."""
    try:
        client.connect()
        print("Connected to MQTT broker.")
    except Exception as e:
        print("Failed to connect to MQTT broker:", e)

def disconnect_mqtt():
    client.disconnect()
    print("Disconnected from MQTT broker.")

def publish_sensor_data(lumi, temp, humi ):
    """Publish sensor data to MQTT broker."""
    if lumi is not None and temp is not None and humi is not None:
        message = {
```

```
            "lumi": lumi,
            "temp": temp,
            "humi": humi
        }
        client.publish(MQTT_TOPIC, str(message))
        print("Published:", message)
    else:
        print("Failed to publish sensor data.")
# --- Receive LoRa Packet ---
def onReceive(lora_modem,payload):
    rssi = lora_modem.packetRssi()
    if len(payload)==32:
        rssi = lora_modem.packetRssi()
        data=aes_decrypt(payload,AES_KEY)
        chan, wkey, lumi, temp, humi = ustruct.unpack('i16s3f', data)
        print("Received encrypted LoRa packet with RSSI: "+str(rssi))   #, payload.decode())
        print(chan,wkey,lumi,temp,humi)
        sensors_display(8,9,lumi,temp,humi,0)
        connect_mqtt()
        publish_sensor_data(lumi, temp, humi)
        disconnect_mqtt()
        rcycle=random.randint(5,15)
        ack=ustruct.pack('2ifi',chan,rcycle,0.01,10)  # chan,cycle, delta, kpack
        enc_ack=aes_encrypt(ack,AES_KEY)
        lora_modem.println(enc_ack)  # sending ACK packet
        print("send encrypted ack AES packet")
        lora_modem.receive()

def main():
    lora_modem.onReceive(onReceive)
    lora_modem.receive()
    if connect_WiFi(SSID,PASS):
        print("WiFi connected")
    while True:
        time.sleep(2)
        print("in the loop")

main()
```

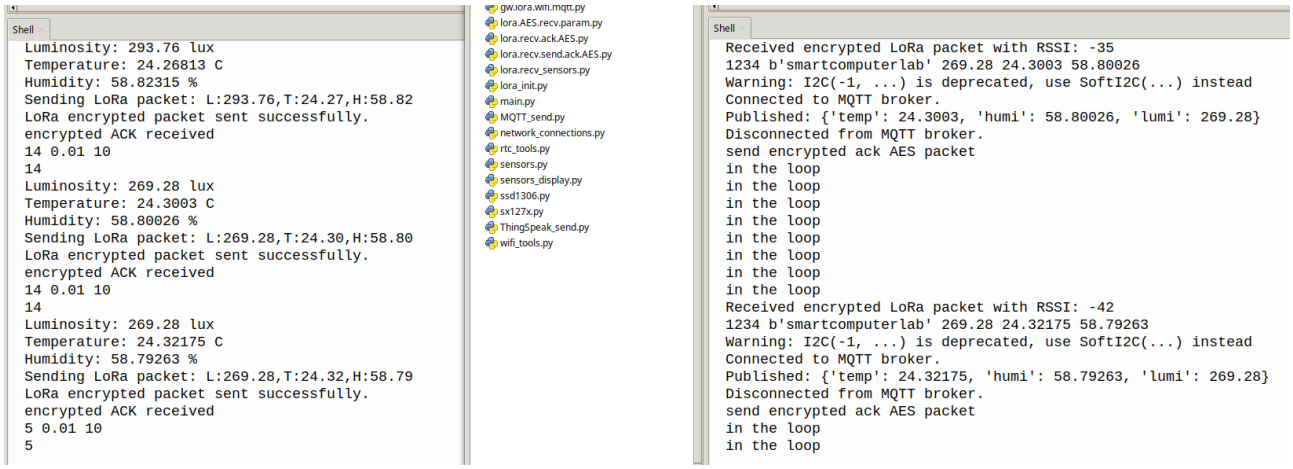

Fig 3.9 LoRa sender and receiver gateway (LoRa-WiFi) to MQTT broker.

To do

Analyze the code and run it with your credentials: WiFi and MQTT broker.

You can also run mosquitto_sub program on your PC/SBC and observe the published messages.

```
bako@bako-U820:~$ mosquitto_sub -h broker.emqx.io -t esp32/sensor_data
{'temp': 24.43973, 'humi': 58.62479, 'lumi': 220.32}
{'temp': 24.43973, 'humi': 58.40353, 'lumi': 220.32}
{'temp': 24.40755, 'humi': 58.43405, 'lumi': 220.32}
```

3.3.3 LoRa-WiFi gateway to ThingSpeak server

```
-----------------------------------------------------------------------------------------------------
from machine import Pin, I2C, SPI
import ustruct, random, ubinascii, urequests
from lora_init import *
from display_sensors import *
from aes_tools import *
import machine,time
from wifi_tools import *

# WiFi credentials
SSID = 'Bbox-9ECEBF79'
PASS = '54347A3EA6A1D6C36EF6A9E5156F7D'

AES_KEY = b'smartcomputerlab'  # Replace with your actual 16-byte AES key
# Initialize LoRa modem
lora_modem = lora_init()
rssi=0; chan=0; wkey=""; lumi=0.0; temp=0.0; humi=0.0

 # Function to send data to ThingSpeak
def send_data_to_thingspeak(lumi, temp, humi, rssi):
    try:
        sf1="&field1="+str(lumi); sf2="&field2="+str(temp); sf3="&field3="+str(humi);
sf4="&field4="+str(rssi)
        url = "https://thingspeak.com/update?key=YOX31M0EDKO0JATK"+sf1+sf2+sf3+sf4
        response = urequests.get(url)
        response.close()
        print("Data sent to ThingSpeak:", lumi, temp, humi, rssi)
    except Exception as e:
        print("Failed to send data:", e)

# --- Receive LoRa Packet ---
def onReceive(lora_modem,payload):
    global rssi; global chan; global wkey; global lumi; global temp; global humi
    rssi = lora_modem.packetRssi()
    if len(payload)==32:
        rssi = lora_modem.packetRssi()
        data=aes_decrypt(payload,AES_KEY)
        chan, wkey, lumi, temp, humi = ustruct.unpack('i16s3f', data)
        print("Received encrypted LoRa packet with RSSI: "+str(rssi))   #, payload.decode())
        print(chan,wkey,lumi,temp,humi)
        display_sensors(8,9,lumi,temp,humi,0)
        rcycle=random.randint(5,15)
        ack=ustruct.pack('2ifi',chan,rcycle,0.01,10)  # chan,cycle, delta, kpack
        enc_ack=aes_encrypt(ack,AES_KEY)
        lora_modem.println(enc_ack)  # sending ACK packet
        print("send encrypted ack AES packet")
        lora_modem.receive()


def main():
    global rssi; global lumi; global temp; global humi
    lora_modem.onReceive(onReceive)
    lora_modem.receive()
    while True:
        if connect_WiFi(SSID, PASS):
            print("WiFi connected")
        send_data_to_thingspeak(lumi, temp, humi, rssi)
        time.sleep(1)
        disconnect_WiFi()
        time.sleep(15)


main()
-----------------------------------------------------------------------------------------------------
```

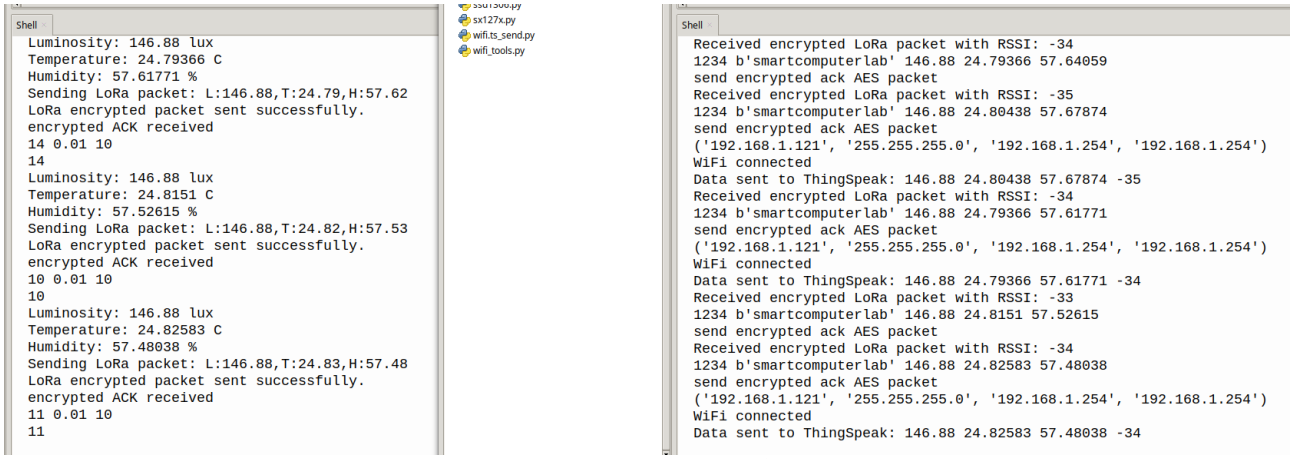

Fig 3.10 LoRa sender and receiver gateway (LoRa-WiFi) to ThingSpeak server.

To do

Test the sender and gateway with your credentials for WiFi and ThingSpeak

---

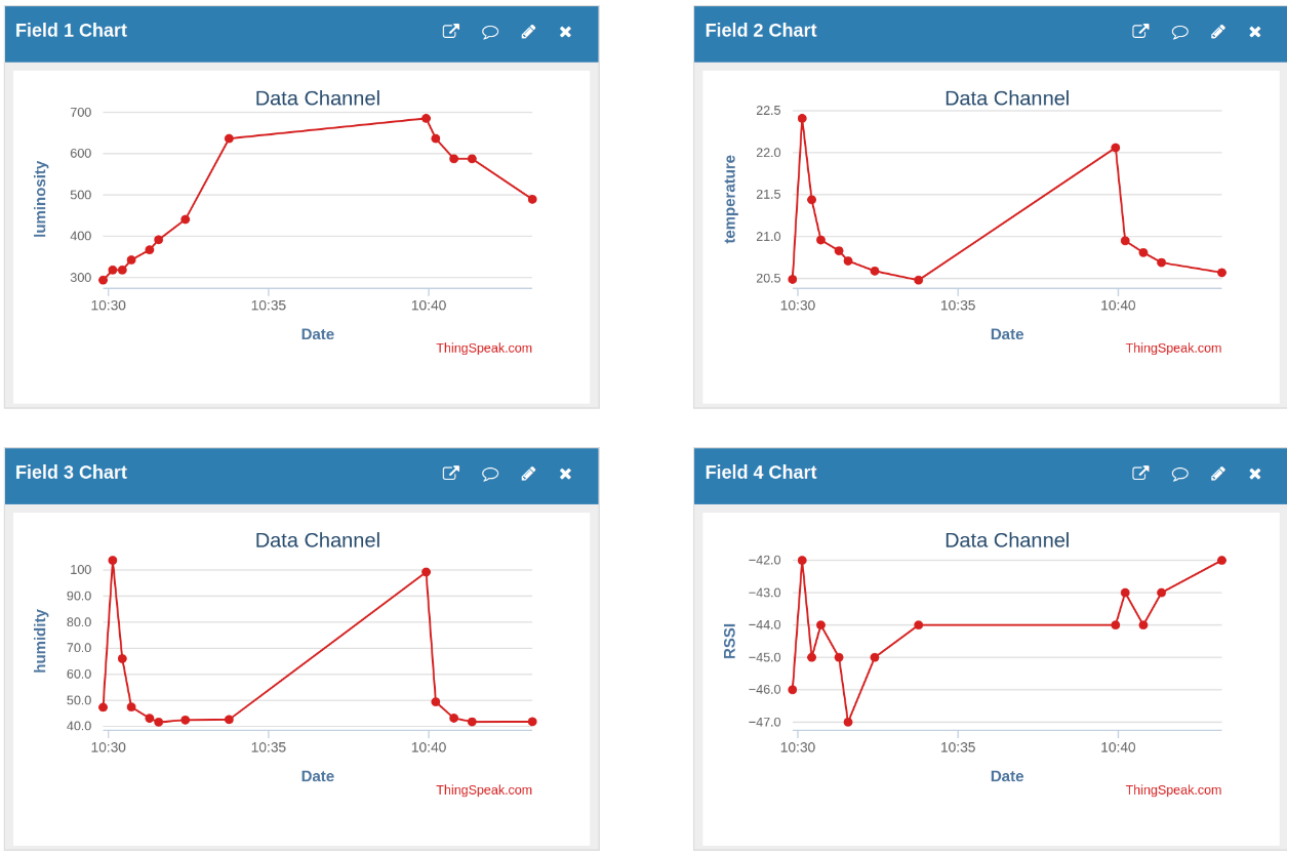

Fig 3.11 ThingSpeak diagrams showing the impact of delta parameters on the sending sequence (stream)