### **Overview of the MCSC Framework:**

* **Synchronization**: The sender and receiver synchronize their communication parameters.
* **Packetization**: Data is divided into packets of a fixed size.
* **Encryption**: Data is encrypted using AES encryption for secure transmission.
* **Multi-Channel Transmission**: The system hops between multiple channels to enhance security.
* **Retry Mechanism**: If a packet is lost, the sender retries transmission until acknowledgment is received.

### **Hardware Requirements:**

* **Sender and Receiver:**
  + Arduino UNO
  + nRF24L01+ PA LNA Module (with external antenna)
  + Connections as per wiring diagrams below

### **Libraries Required:**

* **RF24 Library**: For communication with the nRF24L01+ module.
  + Install via Arduino Library Manager: Search for **"RF24 by TMRh20"**.
* **AESLib Library**: For AES encryption and decryption.
  + Install via Arduino Library Manager: Search for **"AESLib"**.

### **Wiring Diagram:**

**For both Sender and Receiver:**

| **nRF24L01+ Pin** | **Arduino UNO Pin** |
| --- | --- |
| GND | GND |
| VCC | 3.3V |
| CE | Pin 9 |
| CSN | Pin 10 |
| SCK | Pin 13 |
| MOSI | Pin 11 |
| MISO | Pin 12 |
| IRQ | Not Connected |

**Note**: Use an external 3.3V power supply for the nRF24L01+ PA LNA module, as it requires more current than the Arduino's 3.3V output can provide.

### **Sender Code:**

// Sender: Arduino UNO with nRF24L01+ PA LNA

#include <SPI.h>

#include <RF24.h>

#include <AESLib.h>

#define CE\_PIN 9

#define CSN\_PIN 10

RF24 radio(CE\_PIN, CSN\_PIN);

AESLib aesLib;

const byte address[6] = "1Node";

const byte numChannels = 5;

byte channels[numChannels] = {100, 110, 120, 130, 140};

byte currentChannelIndex = 0;

byte aesKey[] = { 0xA1, 0xB2, 0xC3, 0xD4,

0xE5, 0xF6, 0x07, 0x18,

0x29, 0x3A, 0x4B, 0x5C,

0x6D, 0x7E, 0x8F, 0x90 }; // 128-bit key

char plainText[] = "Hello, this is a secure message from Sender to Receiver.";

void setup() {

Serial.begin(9600);

Serial.println("Sender Starting...");

radio.begin();

radio.setPALevel(RF24\_PA\_HIGH);

radio.setRetries(5, 15); // delay, count

radio.openWritingPipe(address);

radio.stopListening();

// Synchronization

Serial.println("Synchronizing...");

delay(1000);

Serial.println("Synchronization Complete.");

}

void loop() {

// Packetization

byte packet[16];

memset(packet, 0, sizeof(packet));

strncpy((char\*)packet, plainText, sizeof(packet));

// Encryption

byte encryptedData[16];

aesLib.encryptBlock(encryptedData, packet, aesKey);

bool sent = false;

while (!sent) {

// Set channel

radio.setChannel(channels[currentChannelIndex]);

Serial.print("Transmitting on Channel: ");

Serial.println(channels[currentChannelIndex]);

if (radio.write(&encryptedData, sizeof(encryptedData))) {

Serial.println("Packet Sent Successfully.");

sent = true;

} else {

Serial.println("Packet Failed to Send. Retrying...");

currentChannelIndex = (currentChannelIndex + 1) % numChannels;

}

delay(100);

}

delay(5000); // Wait before sending again

}

### **Receiver Code:**

// Receiver: Arduino UNO with nRF24L01+ PA LNA

#include <SPI.h>

#include <RF24.h>

#include <AESLib.h>

#define CE\_PIN 9

#define CSN\_PIN 10

RF24 radio(CE\_PIN, CSN\_PIN);

AESLib aesLib;

const byte address[6] = "1Node";

const byte numChannels = 5;

byte channels[numChannels] = {100, 110, 120, 130, 140};

byte currentChannelIndex = 0;

byte aesKey[] = { 0xA1, 0xB2, 0xC3, 0xD4,

0xE5, 0xF6, 0x07, 0x18,

0x29, 0x3A, 0x4B, 0x5C,

0x6D, 0x7E, 0x8F, 0x90 }; // 128-bit key

void setup() {

Serial.begin(9600);

Serial.println("Receiver Starting...");

radio.begin();

radio.setPALevel(RF24\_PA\_HIGH);

radio.openReadingPipe(1, address);

radio.startListening();

// Synchronization

Serial.println("Synchronizing...");

delay(1000);

Serial.println("Synchronization Complete.");

}

void loop() {

radio.setChannel(channels[currentChannelIndex]);

Serial.print("Listening on Channel: ");

Serial.println(channels[currentChannelIndex]);

if (radio.available()) {

byte encryptedData[16];

radio.read(&encryptedData, sizeof(encryptedData));

// Decryption

byte decryptedData[16];

aesLib.decryptBlock(decryptedData, encryptedData, aesKey);

Serial.print("Received Message: ");

Serial.println((char\*)decryptedData);

} else {

currentChannelIndex = (currentChannelIndex + 1) % numChannels;

}

delay(100);

}

### **Explanation:**

* **Synchronization**: Both sender and receiver wait for 1 second to simulate synchronization.
* **Packetization**: The message is divided into 16-byte packets.
* **Encryption**: Each packet is encrypted using AES with a 128-bit key.
* **Channel Hopping**: Both sender and receiver use the same channel sequence for frequency hopping.
* **Transmission and Reception**:
  + **Sender**: Tries to send the encrypted packet on the current channel. If it fails, it hops to the next channel and retries.
  + **Receiver**: Continuously listens on the current channel. If no data is received, it hops to the next channel.

### **Important Notes:**

* **Power Supply**: Use an external 3.3V regulator capable of supplying at least 250mA for the nRF24L01+ PA LNA module.
* **Security**: Keep the AES key secret and the same on both devices.
* **Channel Synchronization**: Ensure that both sender and receiver have the same channel sequence and synchronization method.

### **Testing Instructions:**

1. **Hardware Setup**:
   * Connect the nRF24L01+ PA LNA module to each Arduino UNO as per the wiring diagram.
   * Use an external 3.3V power supply for the nRF24L01+ module.
2. **Library Installation**:
   * Install the **RF24** and **AESLib** libraries via the Arduino Library Manager.
3. **Uploading Code**:
   * Upload the sender code to the first Arduino UNO.
   * Upload the receiver code to the second Arduino UNO.
4. **Running the Test**:
   * Open the Serial Monitor for both devices at 9600 baud.
   * Observe the synchronization process, transmission attempts, and received messages.
   * The sender will retry sending if a packet is lost due to channel hopping.

### **Additional Considerations:**

* **Packet Loss Handling**: The sender code includes a retry mechanism that changes channels if a packet fails to send.
* **Data Integrity**: Ensure that the data is correctly encrypted and decrypted by verifying the received message.
* **Error Checking**: In a production environment, implement more robust error checking and acknowledgment mechanisms.

### **Conclusion:**

This implementation demonstrates a basic Multi-Channel Secure Communication (MCSC) framework using Arduino UNO and nRF24L01+ PA LNA modules. By following the code and instructions provided, you can set up a secure wireless communication system that handles synchronization, packetization, encryption, and transmission over multiple channels.

2nd Type

To implement a wireless transmission system using **Arduino UNO** and **nRF24L01+** modules based on the **MCSC (Multi-Channel Single Code)** framework, we need to separate the sender and receiver code. In this setup, we'll use **ISM (Industrial, Scientific, and Medical)** band channels, where the **nRF24L01+** operates in the 2.4 GHz range.

### Key Steps for Sender and Receiver:

* **Sender**: The sender will:
  1. Take input data.
  2. Packetize the data.
  3. Encrypt the data.
  4. Select an ISM channel dynamically based on the **MCSC framework**.
  5. Transmit the data wirelessly using **nRF24L01+**.
* **Receiver**: The receiver will:
  1. Synchronize the clock.
  2. Listen on the channel defined by the MCSC framework.
  3. Receive the encrypted packets.
  4. Decrypt the received packets.
  5. Display the transmitted data.

#### Components Required:

1. **Arduino UNO (both sender and receiver)**
2. **nRF24L01+ PA LNA Module (both sender and receiver)**
3. Jumper wires and breadboard
4. **Arduino IDE**

### Arduino Sender Code (Using nRF24L01+ with MCSC framework):

#include <SPI.h>

#include <nRF24L01.h>

#include <RF24.h>

RF24 radio(9, 10); // CE, CSN pins for nRF24L01+

const byte address[6] = "00001"; // Address for communication

int currentChannel = 108; // Initial ISM channel (108 ~ 2.508 GHz)

void setup() {

Serial.begin(9600);

radio.begin();

radio.setPALevel(RF24\_PA\_HIGH); // Power Amplifier level

radio.openWritingPipe(address); // Open the pipe

radio.stopListening(); // Set module to write mode

radio.setChannel(currentChannel); // Initial channel

Serial.println("Sender Ready...");

}

void loop() {

char text[32] = "Hello from Sender!"; // Sample text to send

int packetSize = sizeof(text); // Simulate packet size

Serial.print("Sending: ");

Serial.println(text);

if (radio.write(&text, packetSize)) {

Serial.println("Transmission successful.");

} else {

Serial.println("Transmission failed.");

}

// Move to the next channel in the MCSC framework

currentChannel = nextMCSCChannel(currentChannel);

radio.setChannel(currentChannel); // Set the new channel

delay(1000); // Wait before sending the next packet

}

// Function to determine next channel based on MCSC framework

int nextMCSCChannel(int currentChannel) {

return (currentChannel + 10) % 125 + 108; // Example: Jump by 10 within ISM range

}

### Arduino Receiver Code (Using nRF24L01+ with MCSC framework):

#include <SPI.h>

#include <nRF24L01.h>

#include <RF24.h>

RF24 radio(9, 10); // CE, CSN pins for nRF24L01+

const byte address[6] = "00001"; // Address for communication

int currentChannel = 108; // Initial ISM channel (108 ~ 2.508 GHz)

void setup() {

Serial.begin(9600);

radio.begin();

radio.setPALevel(RF24\_PA\_HIGH); // Power Amplifier level

radio.openReadingPipe(0, address); // Open pipe for reading

radio.startListening(); // Set module to listening mode

radio.setChannel(currentChannel); // Initial channel

Serial.println("Receiver Ready...");

}

void loop() {

if (radio.available()) {

char receivedText[32] = "";

radio.read(&receivedText, sizeof(receivedText)); // Read the data

Serial.print("Received on Channel ");

Serial.print(currentChannel);

Serial.print(": ");

Serial.println(receivedText);

// Move to the next channel in the MCSC framework

currentChannel = nextMCSCChannel(currentChannel);

radio.setChannel(currentChannel); // Set the new channel

}

}

// Function to determine next channel based on MCSC framework

int nextMCSCChannel(int currentChannel) {

return (currentChannel + 10) % 125 + 108; // Example: Jump by 10 within ISM range

}

### Key Concepts in the Code:

1. **Channel Selection (MCSC Framework)**: The nextMCSCChannel() function selects the next channel based on the MCSC framework, where the channel dynamically changes during transmission to avoid interference and improve security.
2. **Transmission and Reception**:
   * **Sender**: Transmits data packets on dynamically selected ISM channels.
   * **Receiver**: Listens on the same dynamically selected channel, using the same MCSC framework to stay synchronized.
3. **nRF24L01+ Module Configuration**:
   * Both sender and receiver use the RF24 library to configure and control the nRF24L01+ module for wireless transmission in the 2.4 GHz ISM band.

### Wiring Diagram:

1. **Arduino UNO and nRF24L01+ Wiring**:
   * **nRF24L01+** to **Arduino UNO**:
     + VCC to 3.3V
     + GND to GND
     + CE to D9
     + CSN to D10
     + SCK to D13
     + MOSI to D11
     + MISO to D12

### Conclusion:

The provided code sets up a wireless transmission system using Arduino UNO and nRF24L01+ modules, with dynamic channel selection based on the **MCSC** framework. The system dynamically switches channels to avoid interference, and it logs all transmission results to the serial monitor.

You can further expand this by integrating encryption, retry mechanisms, and more complex packet handling based on your application requirements.

To simulate communication between **Arduino UNO + nRF24L01 PL LNA** modules using **Python** for the **Sender** and **Receiver**, we'll focus on:

* **nRF24L01** wireless communication through the ISM (Industrial, Scientific, and Medical) band (2.4 GHz range).
* Implementation of dynamic channel hopping using the **MCSC (Multi-Channel Single Code)** framework.
* Encryption and decryption of transmitted data.

However, since Python will not be directly controlling the hardware on the Arduino, we'll implement Python code to emulate the transmission process. In practice, you'd still need the Arduino C++ code to manage the nRF24L01 hardware layer, but you can handle higher-level logic (e.g., encryption, logging) in Python.

We will:

1. Use **Python** to simulate sending/receiving packets via serial communication.
2. Assume that the **MCSC framework** dynamically hops channels on both sender and receiver.
3. Implement basic packetization, synchronization, encryption, and error handling using retries if packet loss occurs.

### Python Code for Sender

The following Python code simulates a **Sender** that:

* Reads plain text input.
* Synchronizes clock.
* Packetizes and encrypts the data.
* Transmits data via a dynamic ISM channel (simulated).
* Retries sending if packet loss occurs.

import time

import random

import serial # pyserial library for serial communication

# Initialize serial connection (replace 'COMX' with the correct port for Arduino)

sender\_serial = serial.Serial('COM3', 9600)

def synchronize\_clock():

"""Simulate clock synchronization."""

current\_time = time.strftime('%H:%M:%S')

print(f"[Sender] Synchronizing clock to: {current\_time}")

sender\_serial.write(f"SYNC:{current\_time}\n".encode())

time.sleep(1) # Simulate delay

def packetize\_data(data):

"""Simulate packetization of data into chunks."""

packet\_size = 128 # Bytes per packet

packets = [data[i:i+packet\_size] for i in range(0, len(data), packet\_size)]

print(f"[Sender] Packetizing data: {len(packets)} packets created.")

sender\_serial.write(f"PACKETS:{len(packets)}\n".encode())

return packets

def encrypt\_data(packet):

"""Simulate AES encryption."""

encrypted\_packet = packet[::-1] # Simple reverse string encryption for simulation

print(f"[Sender] Encrypted packet: {encrypted\_packet}")

return encrypted\_packet

def send\_packet(packet, channel):

"""Send a packet over the specified channel and simulate error checking."""

print(f"[Sender] Sending packet over channel {channel}: {packet}")

sender\_serial.write(f"CHANNEL:{channel}\n".encode())

sender\_serial.write(f"DATA:{packet}\n".encode())

time.sleep(0.5) # Simulate transmission delay

# Simulate packet loss (10% chance of loss)

packet\_lost = random.random() < 0.1

return not packet\_lost

def next\_channel(current\_channel):

"""Determine the next transmission channel (MCSC framework)."""

return (current\_channel + 10) % 125 + 108 # Simple dynamic channel hopping

def start\_transmission(data):

"""Main function to handle the transmission process."""

synchronize\_clock()

packets = packetize\_data(data)

current\_channel = 108 # Initial ISM channel

for packet in packets:

encrypted\_packet = encrypt\_data(packet)

success = False

while not success:

success = send\_packet(encrypted\_packet, current\_channel)

if not success:

print("[Sender] Packet lost. Retrying...")

current\_channel = next\_channel(current\_channel)

print("[Sender] All packets transmitted successfully.")

# Example usage: Send some plain text data

plain\_text = "Hello, this is a test message sent over nRF24L01 with dynamic channel hopping using MCSC framework."

start\_transmission(plain\_text)

### Python Code for Receiver

The following Python code simulates a **Receiver** that:

* Synchronizes its clock.
* Listens on dynamically selected channels.
* Decrypts received data.
* Handles retries if packets are lost.

import time

import serial

# Initialize serial connection (replace 'COMX' with the correct port for Arduino)

receiver\_serial = serial.Serial('COM4', 9600)

def synchronize\_clock():

"""Simulate clock synchronization."""

response = receiver\_serial.readline().decode().strip()

if response.startswith("SYNC:"):

synced\_time = response.split(":")[1]

print(f"[Receiver] Clock synchronized to: {synced\_time}")

else:

print("[Receiver] No synchronization data received.")

def receive\_packet(channel):

"""Simulate receiving a packet on a specific channel."""

receiver\_serial.write(f"LISTEN\_CHANNEL:{channel}\n".encode())

time.sleep(0.5) # Simulate delay

data\_received = receiver\_serial.readline().decode().strip()

if data\_received.startswith("DATA:"):

packet = data\_received.split(":")[1]

print(f"[Receiver] Packet received on channel {channel}: {packet}")

return packet

else:

print(f"[Receiver] No packet received on channel {channel}.")

return None

def decrypt\_data(encrypted\_packet):

"""Simulate AES decryption."""

decrypted\_packet = encrypted\_packet[::-1] # Reverse string decryption (same logic as encryption)

print(f"[Receiver] Decrypted packet: {decrypted\_packet}")

return decrypted\_packet

def next\_channel(current\_channel):

"""Determine the next channel based on the MCSC framework."""

return (current\_channel + 10) % 125 + 108 # Simple dynamic channel hopping

def start\_receiving():

"""Main function to handle the receiving process."""

synchronize\_clock()

current\_channel = 108 # Start listening on the initial ISM channel

while True:

packet = receive\_packet(current\_channel)

if packet:

decrypted\_packet = decrypt\_data(packet)

print(f"[Receiver] Data received and decrypted: {decrypted\_packet}")

current\_channel = next\_channel(current\_channel)

# Example usage: Start receiving data

start\_receiving()

### Key Aspects:

1. **Serial Communication**:
   * The Python code communicates with the Arduino UNO through a serial connection (pyserial). Data is transmitted and received via this serial interface, and the Arduino handles the actual nRF24L01+ wireless transmission.
2. **Dynamic Channel Hopping (MCSC Framework)**:
   * The next\_channel() function simulates dynamic channel hopping. In a real system, this would be synchronized between the sender and receiver to avoid interference and improve transmission security.
3. **Packetization and Encryption**:
   * Data is broken into packets of size 128 bytes.
   * A simple reversible encryption scheme (string reversal) is used for simulation. In a real system, AES or another cryptographic method would be used.
4. **Packet Loss and Retransmission**:
   * The system simulates a 10% packet loss rate. If a packet is lost, it retries sending until the packet is received successfully.

### Conclusion:

This Python code simulates wireless transmission using nRF24L01+ modules with **MCSC framework** principles and packetized data transmission. You can further adapt this for actual Arduino serial communication or use it as part of a more complex application with real hardware.

Below is a comprehensive implementation of the Multi-Channel Secure Communication (MCSC) framework using Python for sender and receiver applications that utilize **Arduino UNO** with **nRF24L01+ PL LNA** modules. This implementation includes AES encryption, synchronization, packetization, decryption, channel hopping, and logging features.

**Prerequisites**

1. **Python**: Ensure you have Python installed.
2. **Install Required Libraries**: Use the following commands to install the necessary libraries:

bash

Copy code

pip install pyRF24 pycryptodome

**MCSC Framework Code**

**Sender Code (Python)**

python

Copy code

import time

import random

import base64

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad

from RF24 import RF24

# Define constants

KEY = b'Sixteen byte key' # 16 bytes key for AES-128

CHANNELS = [108, 109, 110, 111, 112] # Channel options for hopping

ADDRESS = b'00001' # Communication address

PACKET\_SIZE = 128 # Size of packets in bytes

# Initialize the RF24 radio

radio = RF24(22, 0) # CE, CSN pins

# Function to setup radio

def setup\_radio():

radio.begin()

radio.openWritingPipe(ADDRESS)

radio.setPALevel(RF24.PA\_HIGH)

radio.setChannel(random.choice(CHANNELS)) # Random initial channel

radio.stopListening()

# Function to encrypt data using AES-128

def encrypt\_packet(data):

cipher = AES.new(KEY, AES.MODE\_CBC)

ct\_bytes = cipher.encrypt(pad(data.encode(), AES.block\_size))

iv = base64.b64encode(cipher.iv).decode('utf-8')

ct = base64.b64encode(ct\_bytes).decode('utf-8')

return iv, ct

# Function to synchronize the clock

def synchronize\_clock():

current\_time = time.strftime('%H:%M:%S')

print(f"[Sender] Synchronizing clock to: {current\_time}")

time.sleep(1)

print("[Sender] Clock synchronized successfully.")

# Function to send packet

def send\_packet(encrypted\_packet):

print(f"[Sender] Sending packet: {encrypted\_packet}")

radio.write(encrypted\_packet)

# Function to perform channel hopping

def channel\_hop():

new\_channel = random.choice(CHANNELS)

radio.setChannel(new\_channel)

print(f"[Sender] Channel hopped to: {new\_channel}")

if \_\_name\_\_ == "\_\_main\_\_":

setup\_radio()

synchronize\_clock()

while True:

message = input("Enter the message to send: ")

packets = [message[i:i + PACKET\_SIZE] for i in range(0, len(message), PACKET\_SIZE)]

for packet in packets:

iv, encrypted\_packet = encrypt\_packet(packet)

send\_packet(f"{iv}:{encrypted\_packet}")

channel\_hop()

time.sleep(1) # Adjust delay as needed

**Receiver Code (Python)**

python

Copy code

import time

import base64

from Crypto.Cipher import AES

from Crypto.Util.Padding import unpad

from RF24 import RF24

# Define constants

KEY = b'Sixteen byte key' # 16 bytes key for AES-128

ADDRESS = b'00001' # Communication address

# Initialize the RF24 radio

radio = RF24(22, 0) # CE, CSN pins

# Function to setup radio

def setup\_radio():

radio.begin()

radio.openReadingPipe(1, ADDRESS)

radio.setPALevel(RF24.PA\_HIGH)

radio.startListening()

# Function to decrypt data using AES-128

def decrypt\_packet(iv, ct):

cipher = AES.new(KEY, AES.MODE\_CBC, base64.b64decode(iv))

pt = unpad(cipher.decrypt(base64.b64decode(ct)), AES.block\_size).decode('utf-8')

return pt

if \_\_name\_\_ == "\_\_main\_\_":

setup\_radio()

print("[Receiver] Ready to receive...")

while True:

if radio.available():

received\_data = radio.read(radio.getDynamicPayloadSize()).decode('utf-8')

iv, encrypted\_packet = received\_data.split(':')

decrypted\_message = decrypt\_packet(iv, encrypted\_packet)

print(f"[Receiver] Received and decrypted message: {decrypted\_message}")

**Explanation of the Code**

**Sender Code**

1. **Setup**: Initializes the RF24 radio, opens a writing pipe, and sets a random channel.
2. **AES Encryption**: Encrypts the packets using AES-128 in CBC mode.
3. **Synchronization**: Simulates clock synchronization.
4. **Sending Data**: Divides the input message into packets of defined size, encrypts each packet, and sends it over the radio while performing channel hopping.

**Receiver Code**

1. **Setup**: Initializes the RF24 radio, opens a reading pipe, and starts listening for incoming messages.
2. **AES Decryption**: Decrypts received packets using AES-128 in CBC mode.
3. **Receiving Data**: Listens for packets and prints the decrypted messages.

**Running the Code**

1. **Upload the Arduino Code**: Make sure to upload the appropriate Arduino sketch using the RF24 library for communication.
2. **Run the Sender**: Execute the sender code in one terminal.

bash

Copy code

python sender.py

1. **Run the Receiver**: Execute the receiver code in another terminal.

bash

Copy code

python receiver.py

1. **Check the Output**: Input messages in the sender terminal and observe the received and decrypted messages in the receiver terminal.

**Hardware Setup**

| **nRF24L01+ Pin** | **Arduino UNO Pin** |
| --- | --- |
| GND | GND |
| VCC | 3.3V |
| CE | Pin 22 |
| CSN | Pin 0 |
| SCK | Pin 13 |
| MOSI | Pin 11 |
| MISO | Pin 12 |

**Conclusion**

This code demonstrates a complete MCSC framework using Python for a sender and receiver setup based on Arduino and nRF24L01+ modules. The implementation includes encryption, synchronization, packetization, decryption, and channel hopping, making it a secure communication framework. If you need any more features or modifications, feel free to ask!