TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

SmartBreeze

Team Members:

- Corcheș Adrian Stelian
- Ilea Cosmin-Ionuț
- Haragîș Horea

Course: Building Automation

Coordinator: Dr. Ing. Ioan Valentin Sita

Contents

1. Introduction

This project is an Internet of Things (IoT) system designed to remotely control a standard infrared (IR) air conditioner using an Android mobile application and an ESP32 microcontroller. The main objective is to replicate the functions of a traditional AC remote—such as power toggling, temperature adjustments, fan speed control, and mode switching—via a wireless connection over Wi-Fi.

The Android application provides a graphical interface for user interaction, while the ESP32 handles network communication through a WebSocket server and emits corresponding IR signals to the air conditioner. This system offers a low-cost, efficient, and extensible solution for smart home automation.

2. System Overview

The system comprises three main components:

- An Android mobile application built using Jetpack Compose and Kotlin.
- An ESP32 microcontroller programmed with the Arduino framework, running a WebSocket server.
- A standard IR-controlled air conditioner, which receives and responds to signals emitted by the ESP32.

Communication between the Android device and ESP32 is achieved over a local Wi-Fi network using WebSockets. Commands from the app are parsed by the ESP32 and translated into IR signals that are recognized by the AC unit.

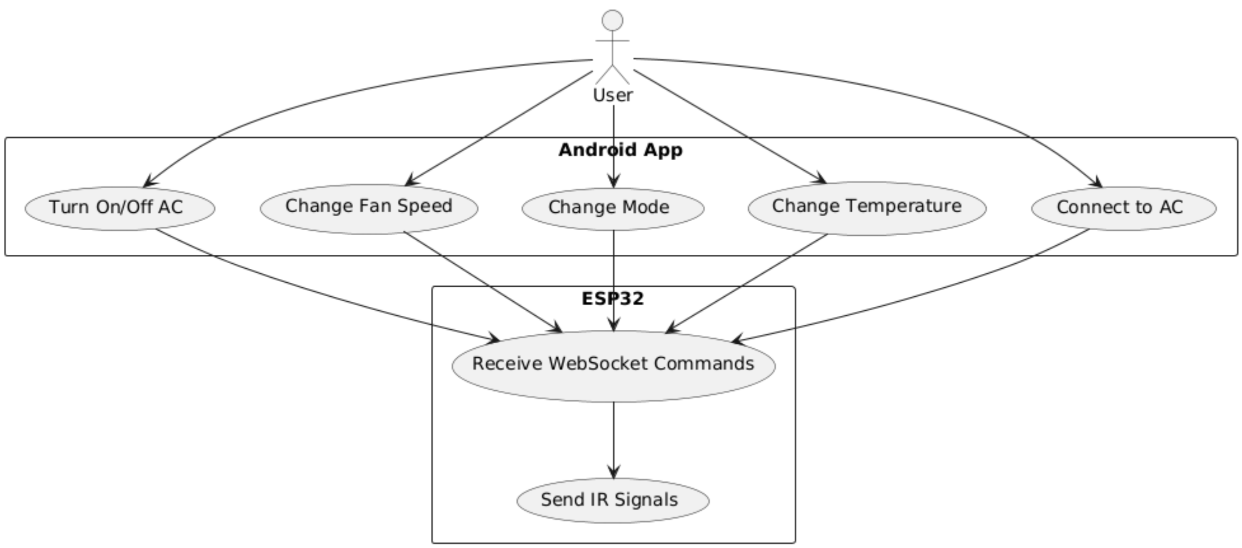
3. Technologies Used

- **Android:** Kotlin, Jetpack Compose, Material 3 Design
- **ESP32:** Arduino C++, AsyncTCP, ESPAsyncWebServer, WebSocketsServer, IRremoteESP8266
- **Networking:** WebSocket protocol over Wi-Fi
- **Hardware:** ESP32 Dev Board, IR LED, IR-controlled Air Conditioner

4. Functional Requirements

- Connect to the ESP32 WebSocket server via the Android app.
- Send IR commands to the air conditioner to:
    - Turn ON/OFF
    - Increase/Decrease temperature
    - Switch between modes (Cool/Heat)
    - Adjust fan speed levels

5. Use Case Diagram



6. System Design

Sequence of Operations:

1. User opens the Android app and taps "Connect to AC".
2. App initiates WebSocket connection to ESP32.
3. ESP32 confirms connection with a status message.
4. User interacts with controls (temperature, mode, fan speed, power).
5. App sends respective commands via WebSocket.
6. ESP32 interprets and emits IR signals.

7. Air conditioner receives and executes the commands.

7. Component Descriptions

Android App

- Built with Jetpack Compose for a responsive UI.
- Maintains state for connection status, temperature, fan speed, and mode.
- Sends formatted WebSocket commands to the ESP32.

ESP32

- Hosts a WebSocket server on port 81.
- Simulates or stores IR codes based on messages received.
- Emits IR signals using an IR LED connected to a GPIO pin.

Air Conditioner

- Standard IR-based AC unit.
- Acts as a passive receiver of IR signals sent from ESP32.

8. Implementation Details

The system is implemented in two layers: the mobile frontend built with Android and the embedded backend running on the ESP32 microcontroller.

ESP32 (C++ / Arduino Framework)

- The ESP32 is programmed using the Arduino IDE and uses the **IRremoteESP8266** library, specifically the IRCoolixAC class, which handles all IR signal formatting compatible with Coolix-based AC units.
- WebSocket communication is implemented using the **AsyncTCP** and **ESPAsyncWebServer** libraries.
- The ESP32 sets up a **WebSocket server on port 81** and listens for string-based commands from the Android application.
- Upon receiving a command, the ESP32 interprets it and takes appropriate action:
  - SEND:ON / SEND:OFF: Turns the air conditioner on or off.

- SEND:TEMP_UP / SEND:TEMP_DOWN: Adjusts the temperature if within allowed range (kCoolixTempMin to kCoolixTempMax).
- SEND:FAN_X: Sets fan speed where X = 0 (Auto), 1 (Low), 2 (Medium), 3 (High).
- SEND:MODE_COOL or SEND:MODE_HEAT: Changes the operation mode.

- The ESP32 also provides feedback for every action back to the app via WebSocket responses like SENT:TEMP_UP or ERROR:INVALID_MODE_CMD.

a. Wi-Fi & WebSocket Initialization

```
const char* ssid = "UTCN-Guest";
const char* password = "utcluj.ro";

AsyncWebServer server(80);
WebSocketsServer webSocket(81);

void setup() {
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
    }
    webSocket.begin();
    webSocket.onEvent(webSocketEvent);
}
```

b. IR Sender Setup

```
#include <IRremoteESP8266.h>
#include <IRsend.h>
#include <ir_Coolix.h>

#define IR_LED_PIN 22
IRCoolixAC ac(IR_LED_PIN);

ac.begin();
ac.setMode(kCoolixCool);
ac.setTemp(20);
ac.setFan(kCoolixFanAuto);
ac.setPower(false);
```

c. Message Parsing and Action Dispatch

```
if (message == "SEND:ON") {
    ac.setPower(true);
    ac.send();
}
else if (message == "SEND:TEMP_UP") {
    ac.setTemp(ac.getTemp() + 1);
    ac.send();
}
else if (message.startsWith("SEND:FAN_")) {
    int fanSetting = message.substring(9).toInt();
    ac.setFan(fanSetting);  // Uses constants like kCoolixFanMin, etc.
    ac.send();
}
```

Android App (Kotlin / Jetpack Compose)

- The Android app is developed using **Jetpack Compose** for a modern, declarative UI.
- It uses the **OkHttp** library to initiate and manage the WebSocket connection to the ESP32.
- The UI maintains live state for temperature, mode, fan speed, and connection status.
- On user input (e.g., pressing a button), the app formats the command and sends it via WebSocket.
- Example messages include:
    - "SEND:TEMP_UP" — when temperature increase button is tapped
    - "SEND:MODE_COOL" — when switching to Cool mode
    - "SEND:FAN_2" — to set medium fan speed
- The app displays status updates based on ESP32 responses, improving feedback and interactivity.

a. WebSocket Initialization

```
private fun connectToWebSocket() {
    val client = OkHttpClient()
    val request = Request.Builder().url("ws://192.168.1.139:81").build()
    webSocket = client.newWebSocket(request, WebSocketHandler())
}
```

b. Sending Commands

```
private fun sendMessage(message: String) {
    if (::webSocket.isInitialized) {
        webSocket.send(message)
    }
}
```

c. UI Integration with Commands

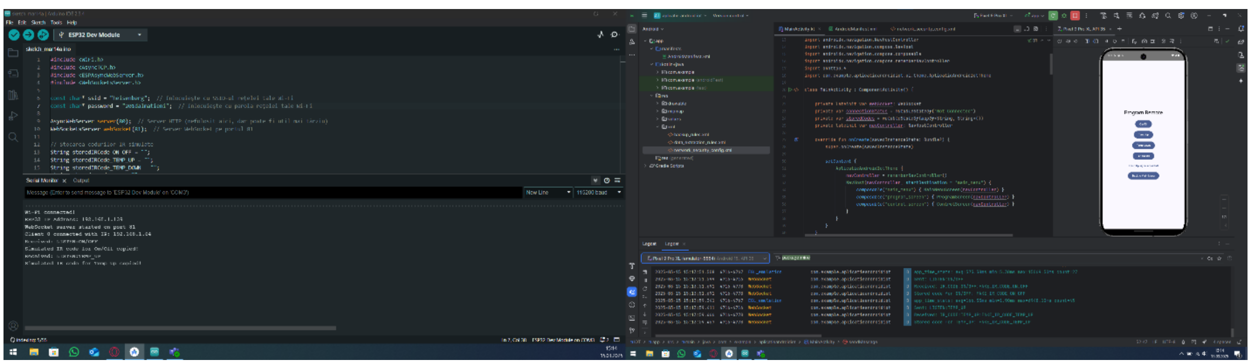
```
Button(onClick = {
    currentTemp++
    sendMessage("SEND:TEMP_UP")
}) {
    Text("+")
}

Button(onClick = {
    fanSpeed = 2
    sendMessage("SEND:FAN_2")
}) {
    Text("Fan Med")
}
```

9. Testing

To validate the communication between the Android app and the ESP32, we performed tests using dummy IR messages. These tests ensured that:

- Commands from the app were correctly formatted and sent over the WebSocket protocol.
- ESP32 successfully received the messages, parsed them, and acknowledged them.
- The serial monitor on the ESP32 displayed confirmation logs such as:
    - Received: LISTEN:ON/OFF
    - Simulated IR code for On/Off copied!
    - Received: LISTEN:TEMP_UP
    - Simulated IR code for Temp Up copied!
- The app logs showed corresponding WebSocket responses from the ESP32.

This confirmed bi-directional communication and correct function triggering on both ends.



10. Challenges and Solutions

- **Challenge:** Ensuring IR codes match the AC brand.
    - **Solution:** Use IRremoteESP8266 with brand-specific protocols.
- **Challenge:** Maintaining stable WebSocket connection.
    - **Solution:** Reconnect logic and connection status display in app.

11. Future Improvements

- Add EEPROM saving for IR codes.
- Support more AC brands and IR protocols.
- Implement voice control (Google Assistant integration).
- Create a scheduling system for automated control.

12. Conclusion

This project successfully demonstrates a working IoT solution for remote air conditioner control using an Android app and ESP32. It merges software, hardware, and networking concepts into a functional and extendable smart home module. With further enhancements, this system can serve as a template for broader smart home automation initiatives.