NATIONAL INSTITUTE OF TECHNOLOGY

KARNATAKA, SURATHKAL

Computer networks

Mini-Project

Application Program for WiFi control of Ceiling Fan with 4 operating speeds as: OFF, LOW, MEDIUM, HIGH.To Incorporate all requirements to comply with the IOT standard prevalent today, Illustrating its complete functioning in simulation mode.

Team members:

Yogesh P (201EE138)

R Geetesh Saravanan (201EE242)

Analysis report:

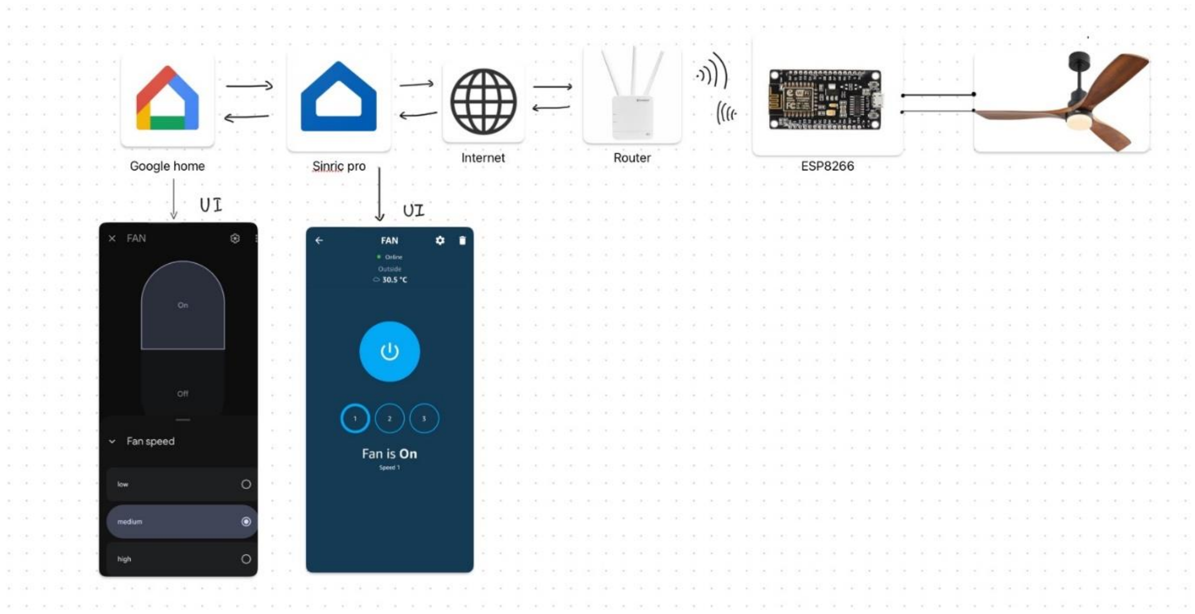


Figure 1:IoT framework with ESP8266 as IoT communicator

Methodology of implementation:

1. Pre-processor Directives:

- The code starts with pre-processor directives to conditionally include libraries based on the target microcontroller (ESP8266, ESP32, or Arduino RP2040).

- It also defines debug configurations for debugging purposes when the ENABLE_DEBUG flag is defined.

2. Library Inclusions:

- The code includes necessary libraries such as Arduino.h , WiFi.h (or ESP8266WiFi.h for ESP8266), and SinricPro.h .

- SinricPro.h is the library for interfacing with the SinricPro IoT platform.

3. Configuration Constants:

- WiFi credentials (WIFI_SSID and WIFI_PASS) are provided for connecting to the local network.
- APP_KEY , APP_SECRET , and FAN_ID are authentication and device identifiers obtained from the SinricPro platform.

4. Device State Struct:

- A struct device_state is defined to store the state and values of the fan.
- It includes powerState to indicate whether the fan is on or off and fanSpeed to represent the speed of the fan.

5. Callback Functions:

- Callback functions onPowerState , onRangeValue , and onAdjustRangeValue are defined to handle events triggered by the SinricPro platform.
- These functions handle power state changes, fan speed changes, and adjustments to the fan speed, respectively.

6. WiFi Setup:

- The setupWiFi() function configures the WiFi connection, attempts to connect to the network, and prints the IP address upon successful connection.

- It also sets up GPIO pin 16 as an output for controlling the fan.

7. SinricPro Setup:

- The setupSinricPro() function initializes the SinricPro library.

- It associates callback functions with the SinricPro fan device (SinricProFanUS) to handle power state changes, fan speed changes, and adjustments.

- Additionally, it sets up callback functions for handling connection and disconnection events with the SinricPro platform.

8. Main Setup Function:

- The setup() function initializes the serial communication and calls setupWiFi() and setupSinricPro() functions.

9. Main Loop:

- The loop() function continuously calls SinricPro.handle() to process incoming requests and maintain communication with the SinricPro platform.

Code Report:

- The code demonstrates how to integrate an ESP8266, ESP32, or Arduino RP2040 device with the SinricPro IoT platform to control a fan remotely.

- It follows a modular approach by defining callback functions to handle different events triggered by the SinricPro platform.

- The setup functions ensure proper initialization of WiFi connection and SinricPro library.

- The main loop continuously handles requests from the SinricPro platform to maintain communication.

- Debugging features are included but can be disabled by defining NDEBUG .

- Overall, the code provides a clear structure for integrating IoT functionality into fan control applications using the SinricPro platform.

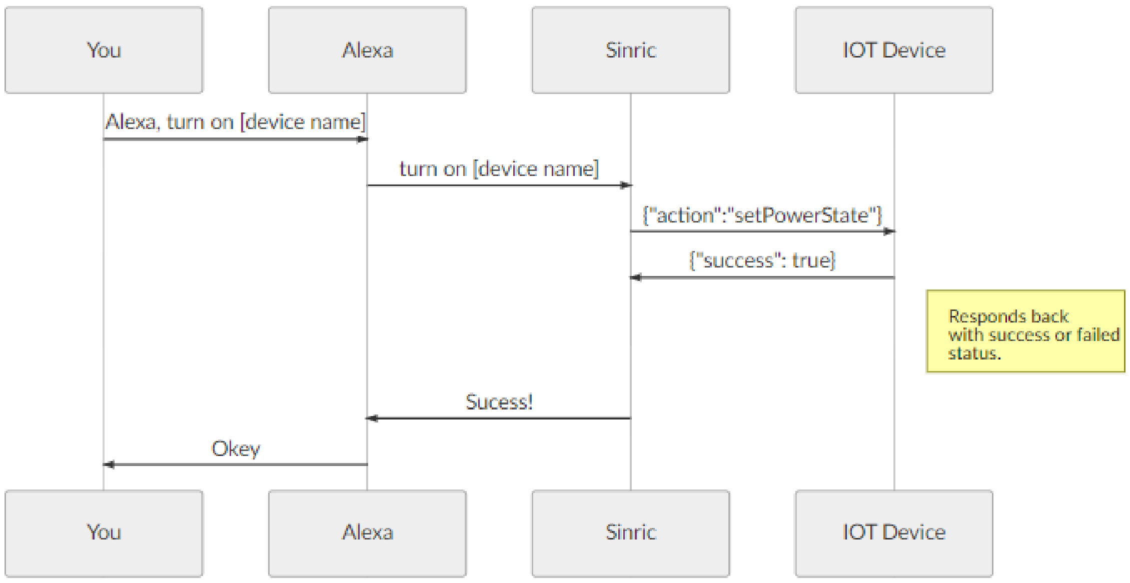
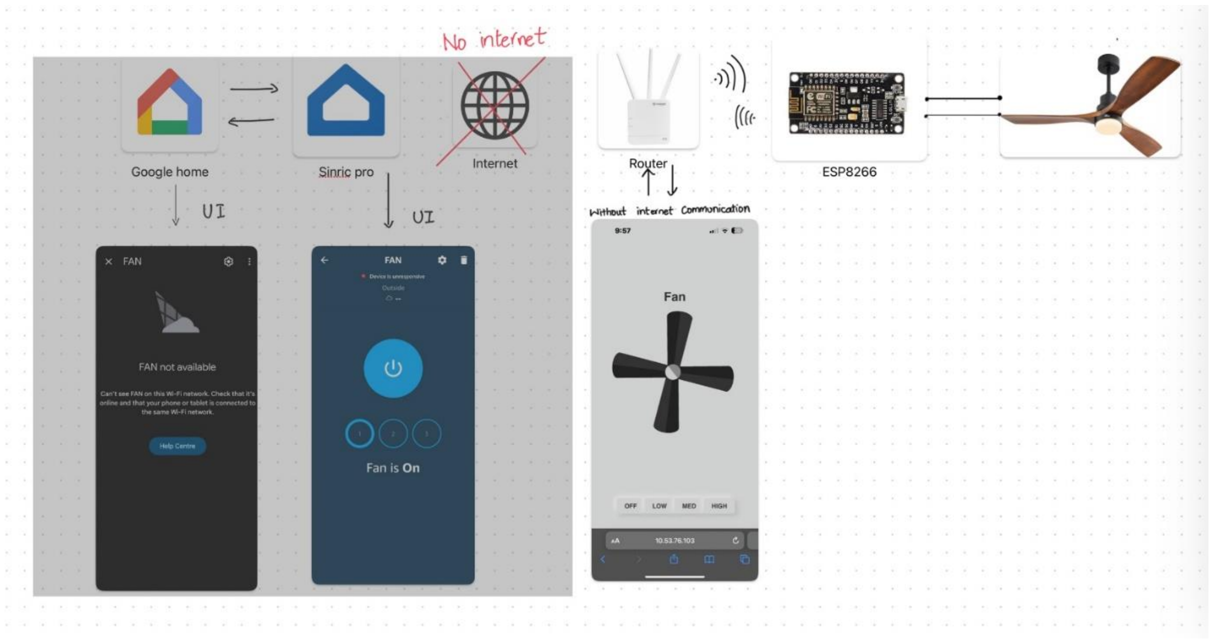

You
Alexa
Sinric
IOT Device
Alexa, turn on [device name]
turn on [device name]
{"action":"setPowerState"}
{"success": true}
Responds back with success or failed status.
Sucess!
Okey
You
Alexa
Sinric
IOT Device

Figure 2:IoT messages



Can't see FAN on this Wi-Fi network. Check that it's online and that your phone or tablet is connected to the same Wi-Fi network.

Figure 3:IoT framework without internet with active communication using LAN

LAN communication:

Methodology of LAN Fan Control System

1. Introduction:

The provided code implements a web-based cooling fan control system using Flask, a micro web framework for Python, and Flask-SocketIO for real-time communication between the client and server. The system allows users to control the speed of a virtual fan through a web interface.

2. Technologies Used:

- Flask: Flask is a micro web framework written in Python. It is lightweight and provides tools and libraries for building web applications.
- Flask-SocketIO: Flask-SocketIO is an extension for Flask that adds support for WebSocket communication. It allows real-time, bi-directional communication between web clients and servers.
- HTML/CSS/JavaScript: Used for building the web interface and controlling the behavior of the fan animation.
- Socket.IO: A JavaScript library for real-time web applications. It enables real-time, event-based communication between the client and server.

3. Components of the System:

Flask Application:

- The Flask application is created with a route / to handle GET and POST requests.
- The GET request serves the HTML template containing the fan control interface.
- The POST request is used to receive fan speed commands from the client.

HTML Template:

- The HTML template renders the fan control interface.
- It contains buttons to control the fan speed and a SVG-based fan animation.
- JavaScript code is embedded in the template to handle button clicks and update the fan animation.

Fan Animation:

- The fan animation is created using SVG and CSS.
- It simulates the rotation of fan blades based on the selected speed.

SocketIO Integration:

- SocketIO is used to emit fan speed update events to all connected clients.

- When a client changes the fan speed, a POST request is sent to the server, which then emits a fan_speed_update event to all clients.

4. Ongoing Process:

- When a user interacts with the web interface by clicking on the speed buttons, JavaScript code sends a POST request to the server with the selected fan speed.

- The server receives the fan speed command, updates the last_fan_speed variable, and emits a fan_speed_update event to all connected clients via SocketIO.

- Clients listening for the fan_speed_update event update the fan animation based on the received fan speed.

5. Conclusion:

The provided code implements a simple yet effective fan control system using Flask and Flask-SocketIO. It demonstrates the integration of server-side logic with real-time communication to provide a responsive web application for controlling a virtual fan.

UI for Local communication

- Many IoT device manufacturers offer dedicated apps for controlling their devices, in addition to compatibility with Google Home, Apple HomeKit, and Alexa.
- However, what's often missing is a communication channel via LAN connection.

  This feature would allow users to control their appliances even when internet access is unavailable, providing a reliable solution for situations where connectivity is limited or disrupted

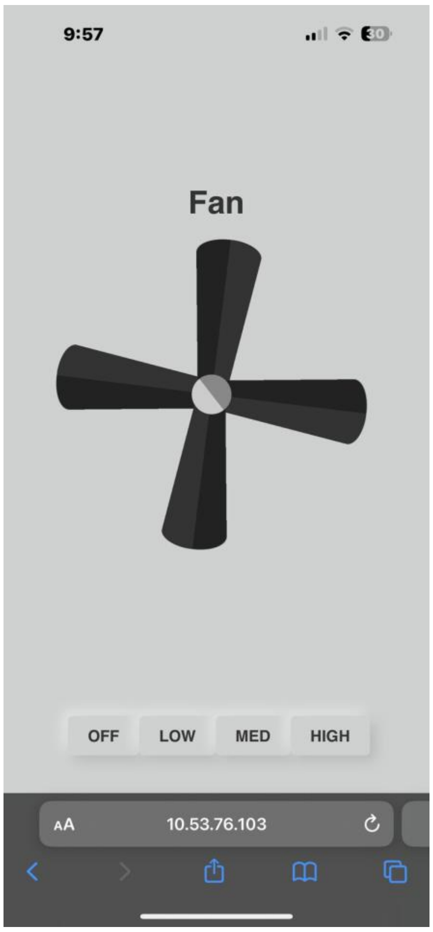


Figure 4: UI provided by the server on smartphone

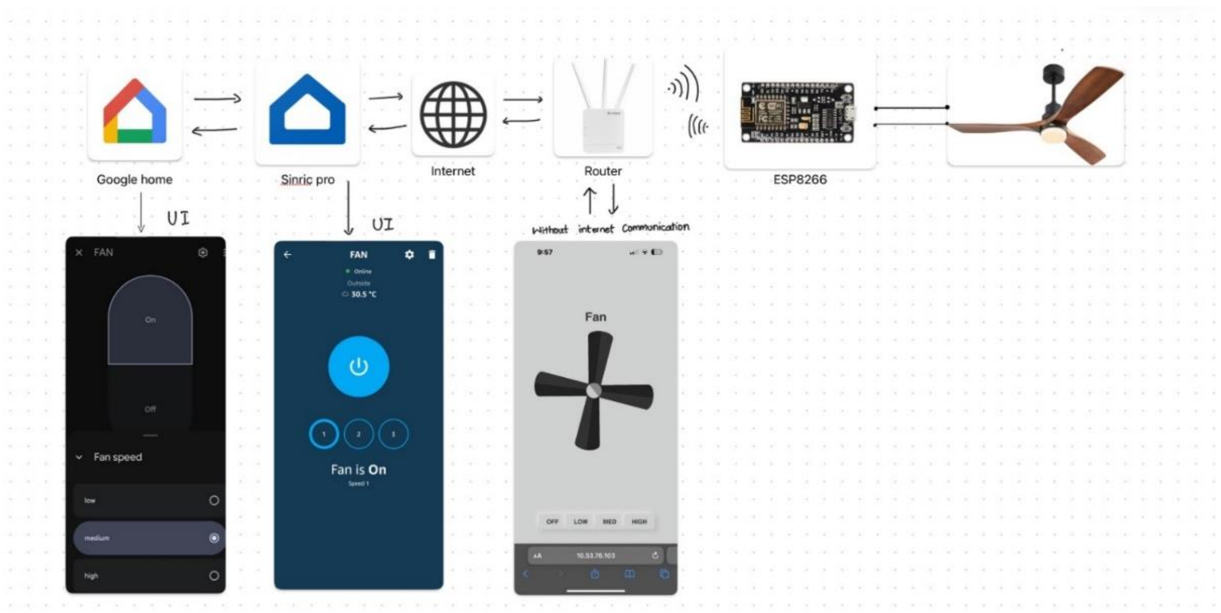
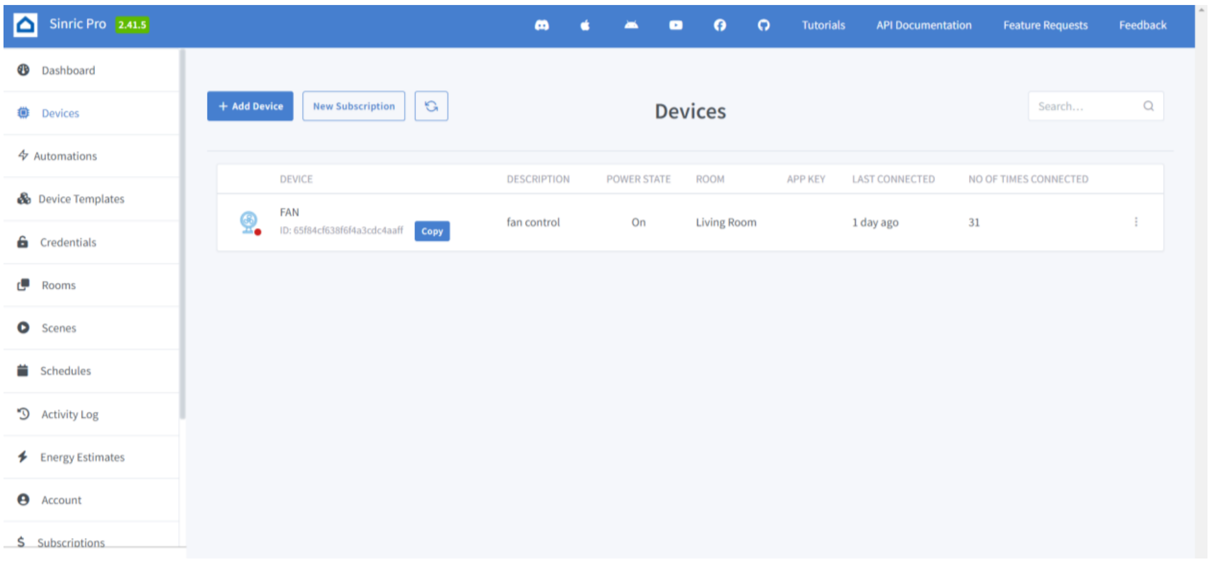


Figure 5:Introducing LAN communication to the IoT framework

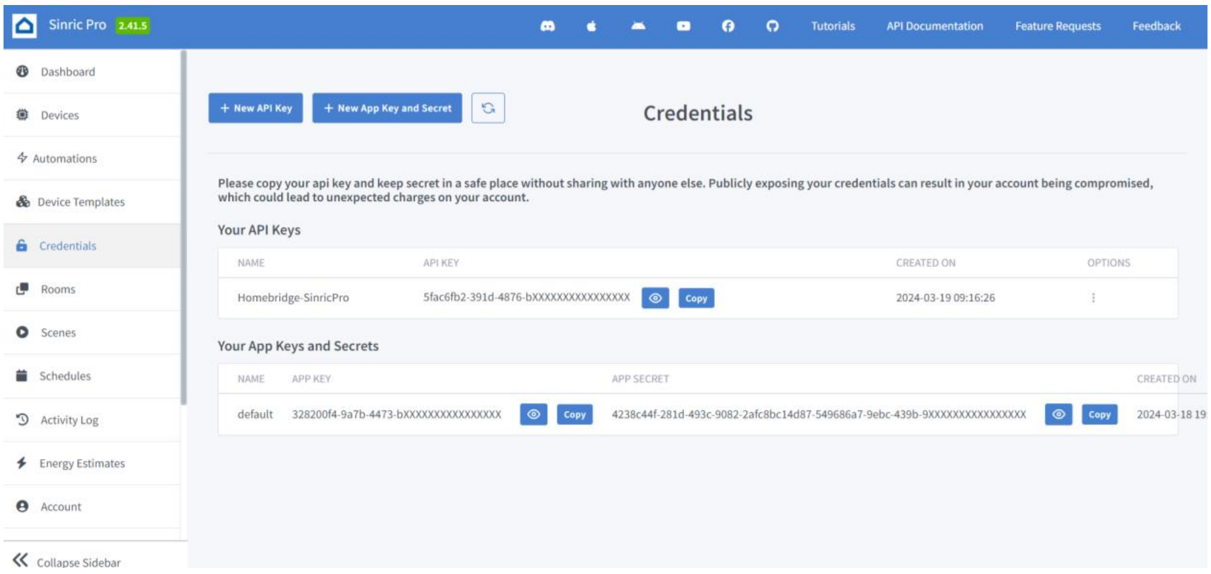
Register on Sinic Pro to obtain API key credentials.



Devices

DEVICE	DESCRIPTION	POWER STATE	ROOM	APP KEY	LAST CONNECTED	NO OF TIMES CONNECTED
FAN ID: 65f84cf638f6f4a3cdc4aaff	fan control	On	Living Room		1 day ago	31

Figure 6Sinicpro device adding page



Credentials

Please copy your api key and keep secret in a safe place without sharing with anyone else. Publicly exposing your credentials can result in your account being compromised, which could lead to unexpected charges on your account.

Your API Keys

NAME	API KEY	CREATED ON	OPTIONS
Homebridge-SinricPro	5fac6fb2-391d-4876-bXXXXXXXXXXXXXXX	2024-03-19 09:16:26	⋮

Your App Keys and Secrets

NAME	APP KEY	APP SECRET	CREATED ON
default	328200f4-9a7b-4473-bXXXXXXXXXXXXXXX	4238c44f-281d-493c-9082-2afc8bc14d87-549686a7-9ebc-439b-9XXXXXXXXXXXXXXX	2024-03-18 19

Figure 7sinicpro credentials page

Results screenshots:

Update the WI-FI network SSID and password, as well as the app key, app secret, and fan ID and transfer the ESP8266_fan.ino code to the device via usb from your laptop.

Install sinic pro app on your phone, you can link it to your google home now by logging in sinic pro to google home

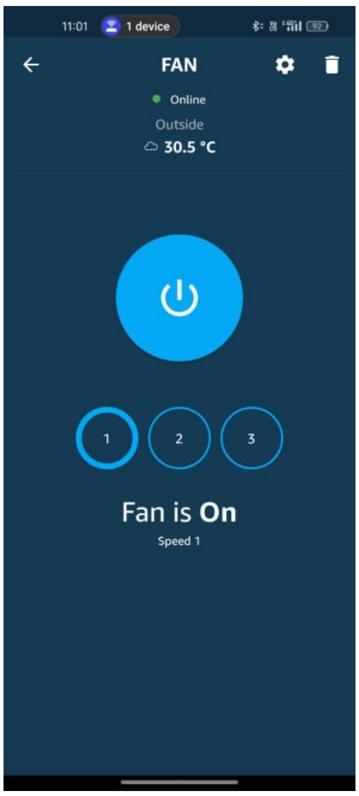


Figure 8: Sinicpro app UI

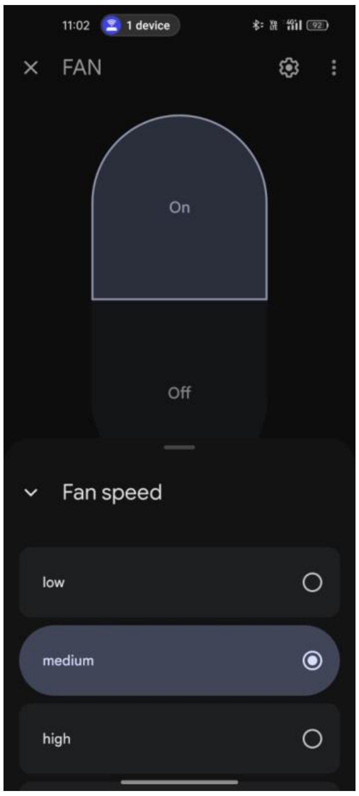
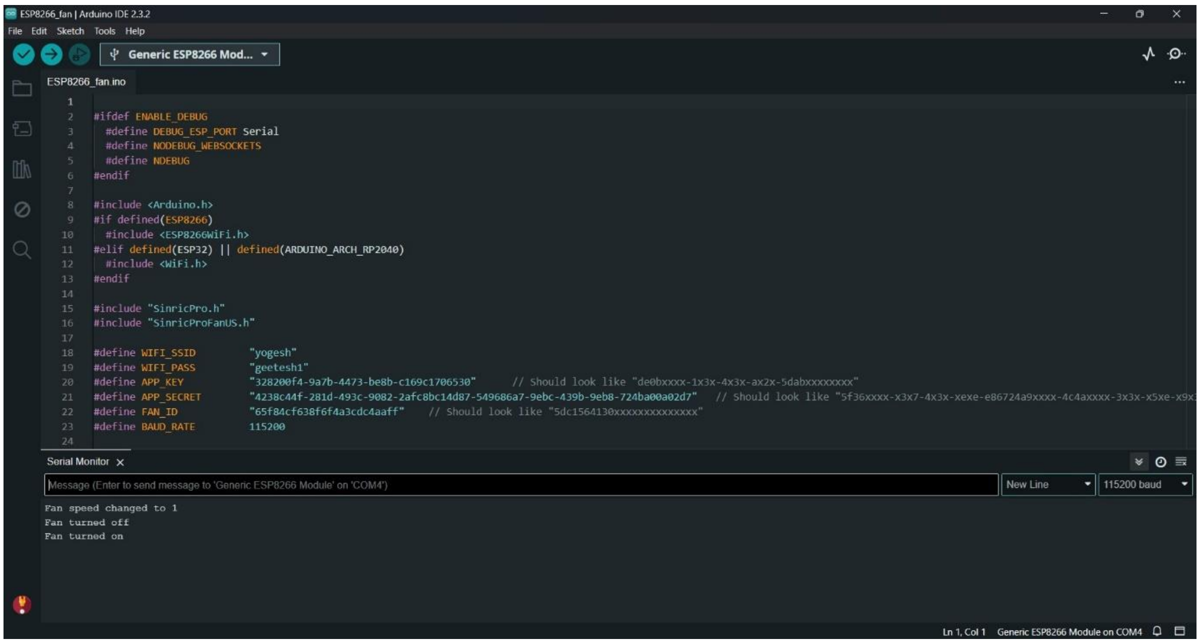


Figure 9: Google Home UI



```

#ifdef ENABLE_DEBUG
  #define DEBUG_ESP_PORT Serial
  #define NODEBUG_WEBSOCKETS
  #define NDEBUG
#endif

#include <Arduino.h>
#if defined(ESP8266)
  #include <ESP8266WiFi.h>
#elif defined(ESP32) || defined(ARDUINO_ARCH_RP2040)
  #include <WiFi.h>
#endif

#include "SinricPro.h"
#include "SinricProFanUS.h"

#define WIFI_SSID         "yogesh"
#define WIFI_PASS         "geetesh1"
#define APP_KEY           "328200f4-9a7b-4473-be8b-c169c1706530"      // Should look like "de0bxxxx-1x3x-4x3x-ax2x-5dabxxxxxxxx"
#define APP_SECRET        "4238c44f-281d-493c-9082-2afc8bc14d87-549686a7-9ebc-439b-9eb8-724ba00a02d7"   // Should look like "5f36xxxx-x3x7-4x3x-xexe-e86724a9xxxx-4c4axxxx-3x3x-x5xe-x9x
#define FAN_ID            "65f84cf638f6f4a3cdc4aaff"    // Should look like "5dc1564130xxxxxxxxxxxxxx"
#define BAUD_RATE         115200

```

Serial Monitor
Message (Enter to send message to 'Generic ESP8266 Module' on 'COM4')
New Line
115200 baud

```
Fan speed changed to 1
Fan turned off
Fan turned on
```

Figure 10: Arduino Serial Monitor with ESP8266 connected to laptop

This code is an Arduino sketch that uses the SinricPro library to control a smart fan device. The fan can be turned on/off, and its speed can be adjusted using voice commands or a mobile app. Here's a flow chart and an explanation of how the different layers of the computer network are used in this application:

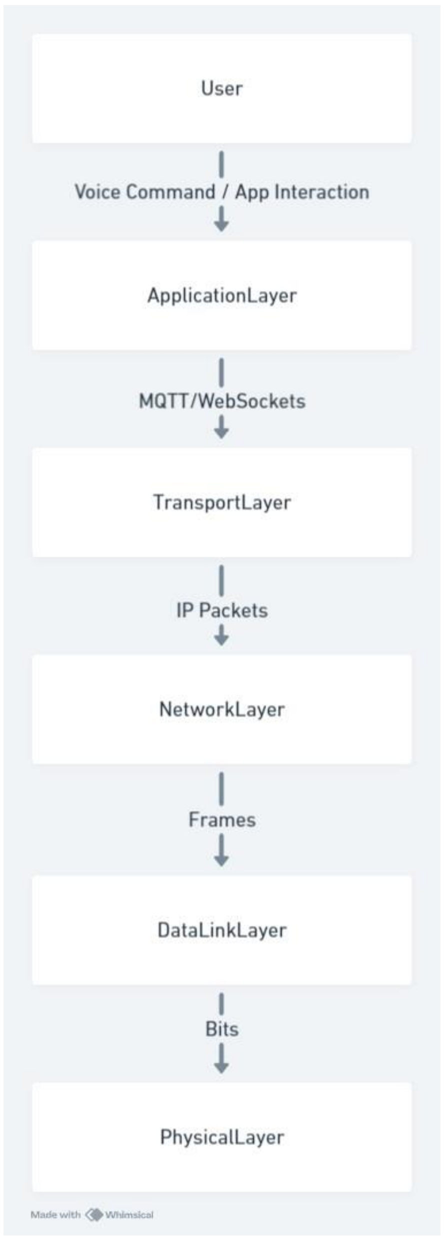


Figure 11: Flow chart for ESP8266 communication

1. Physical Layer: This layer deals with the transmission of raw bits over the physical medium, such as WiFi wireless signals. In this application, it is responsible for transmitting and receiving the data bits between the Arduino device and the SinricPro cloud service.

2. Data Link Layer: This layer handles the reliable transfer of data frames between devices on the same network. In this application, it ensures that the data frames are properly formatted and transmitted between the Arduino device and the WiFi router.

3. Network Layer: This layer is responsible for routing data packets across different networks using IP (Internet Protocol) addresses. In this application, it routes the IP packets containing the MQTT or WebSocket data between the Arduino device and the SinricPro cloud service over the internet.

4. Transport Layer: This layer ensures reliable end-to-end communication between applications running on different devices. In this application, it likely uses TCP (Transmission Control Protocol) to establish a connection between the Arduino device and the SinricPro cloud service, ensuring that the data is delivered reliably and in the correct order.

5. Application Layer: This layer provides protocols and services for applications to communicate with each other. In this application, the MQTT (Message Queuing Telemetry Transport) protocol or WebSockets are used for communication between the Arduino device and the SinricPro cloud service.

When the user interacts with the fan using voice commands or a mobile app, the voice command or app interaction is processed by the SinricPro cloud service. The SinricPro cloud service then sends the appropriate command or data to the Arduino device using MQTT or WebSockets.

The command or data goes through the Application Layer, Transport Layer (TCP), Network Layer (IP), Data Link Layer, and finally, the Physical Layer, where the raw bits are transmitted over the WiFi network.

The Arduino device receives the command or data and processes it using the SinricPro library. It then updates the fan's power state or speed accordingly by controlling the appropriate GPIO pins.

If the fan's state or speed changes, the Arduino device can also send updates back to the SinricPro cloud service through the same layers, but in the reverse order, using MQTT or WebSockets.

The SinricPro cloud service can then update the user interface or provide feedback to the user based on the received updates from the Arduino device

Offline LAN communication Results:

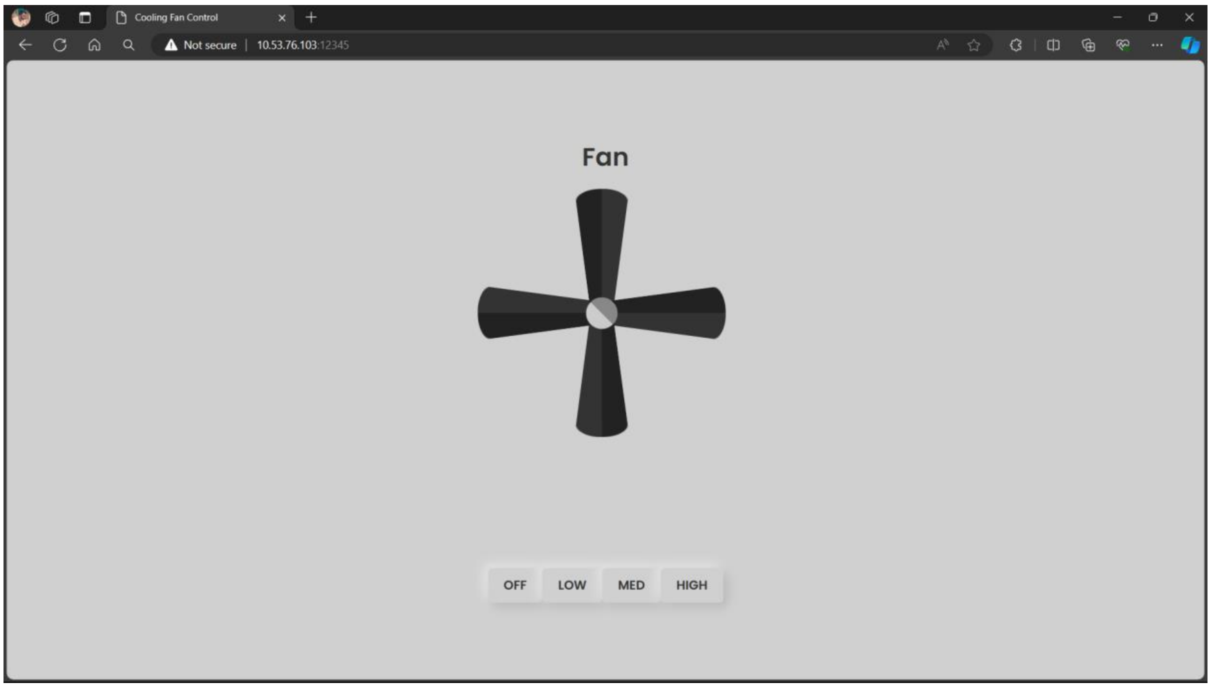
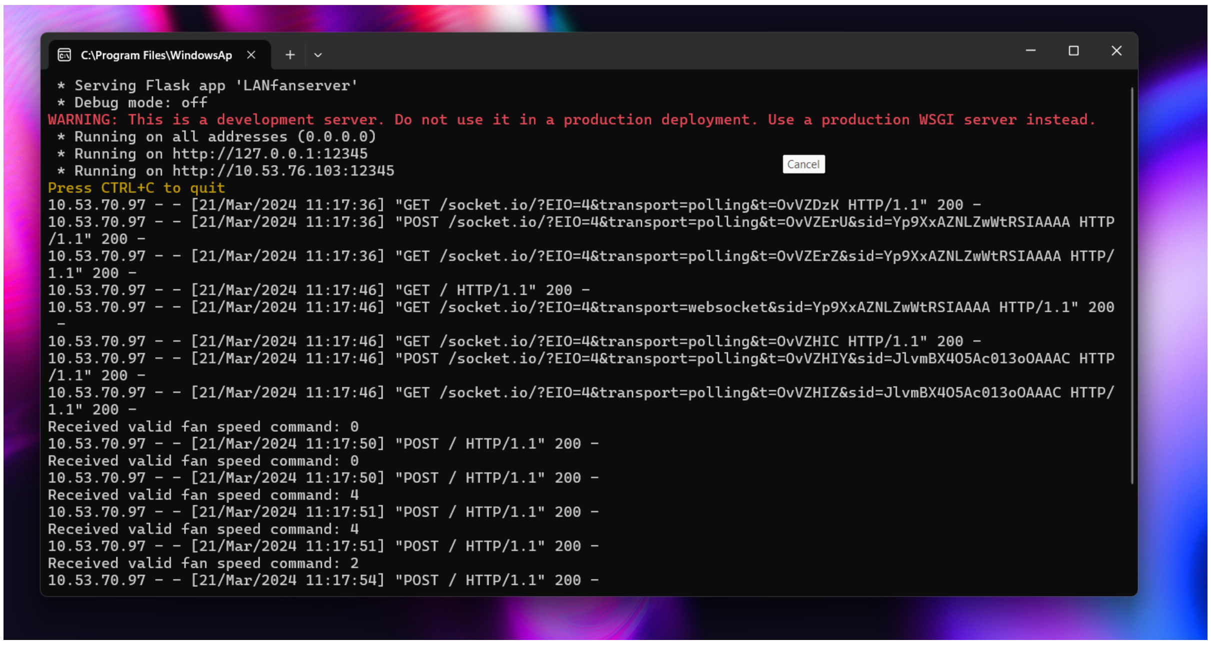


Figure 12:HTML Page served by server in the LAN



```
 * Serving Flask app 'LANfanserver'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:12345
 * Running on http://10.53.76.103:12345
Press CTRL+C to quit
10.53.70.97 - - [21/Mar/2024 11:17:36] "GET /socket.io/?EIO=4&transport=polling&t=OvVZDzK HTTP/1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:36] "POST /socket.io/?EIO=4&transport=polling&t=OvVZErU&sid=Yp9XxAZNLZwWtRSIAAAA HTTP
/1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:36] "GET /socket.io/?EIO=4&transport=polling&t=OvVZErZ&sid=Yp9XxAZNLZwWtRSIAAAA HTTP/
1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:46] "GET / HTTP/1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:46] "GET /socket.io/?EIO=4&transport=websocket&sid=Yp9XxAZNLZwWtRSIAAAA HTTP/1.1" 200
 -
10.53.70.97 - - [21/Mar/2024 11:17:46] "GET /socket.io/?EIO=4&transport=polling&t=OvVZHIC HTTP/1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:46] "POST /socket.io/?EIO=4&transport=polling&t=OvVZHIY&sid=JlvmBX4O5Ac013oOAAAC HTTP
/1.1" 200 -
10.53.70.97 - - [21/Mar/2024 11:17:46] "GET /socket.io/?EIO=4&transport=polling&t=OvVZHIZ&sid=JlvmBX4O5Ac013oOAAAC HTTP/
1.1" 200 -
Received valid fan speed command: 0
10.53.70.97 - - [21/Mar/2024 11:17:50] "POST / HTTP/1.1" 200 -
Received valid fan speed command: 0
10.53.70.97 - - [21/Mar/2024 11:17:50] "POST / HTTP/1.1" 200 -
Received valid fan speed command: 4
10.53.70.97 - - [21/Mar/2024 11:17:51] "POST / HTTP/1.1" 200 -
Received valid fan speed command: 4
10.53.70.97 - - [21/Mar/2024 11:17:51] "POST / HTTP/1.1" 200 -
Received valid fan speed command: 2
10.53.70.97 - - [21/Mar/2024 11:17:54] "POST / HTTP/1.1" 200 -
```

Figure 13: Server in the command center on a different Laptop

This code is a Flask web application that controls the speed of a cooling fan. It uses Flask-SocketIO for real-time communication between the server and the client. Here's a flow chart and an explanation of how the different layers of the computer network are used in this application:

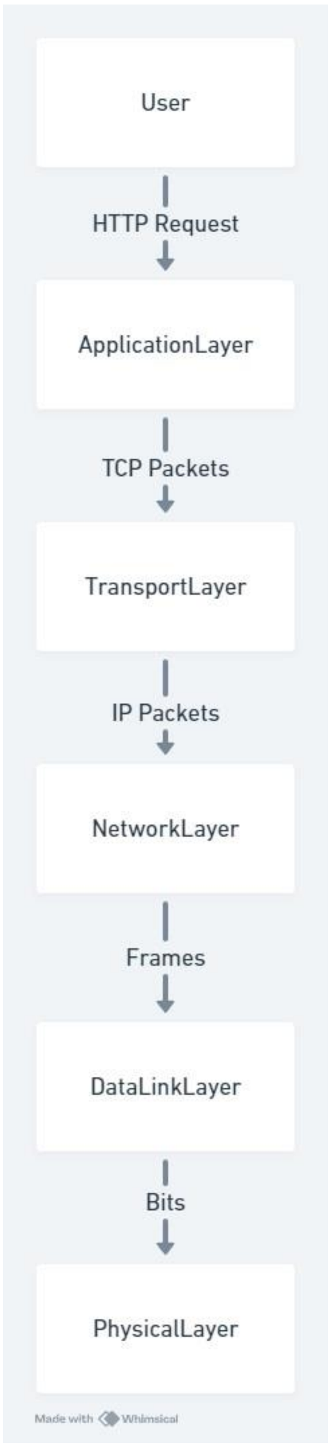


Figure 14: Flow chart for LAN communication

1. Physical Layer: This layer deals with the transmission of raw bits over the physical medium, such as cables or wireless signals. In this application, it is responsible for transmitting and receiving the actual data bits between the client and server.

2. Data Link Layer: This layer handles the reliable transfer of data frames between devices on the same network. It performs tasks like framing, addressing, and error detection/correction. In this application, it ensures that the data frames are properly formatted and transmitted between the client and server without errors.

3. Network Layer: This layer is responsible for routing data packets across different networks. It uses IP (Internet Protocol) addresses to identify the source and destination devices. In this application, the Network Layer routes the IP packets containing the HTTP requests and responses between the client and server over the internet.

4. Transport Layer: This layer ensures reliable end-to-end communication between applications running on different devices. It uses protocols like TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). In this application, TCP is used to establish a connection between the client and server, ensuring that the data is delivered reliably and in the correct order.

5. Application Layer: This layer provides protocols and services for applications to communicate with each other. In this application, the HTTP protocol is used to handle the initial GET and POST requests from the client to the server. Additionally, the application uses WebSockets (through Flask-SocketIO) for real-time communication, enabling the server to send updates about the fan speed to the client.

When the user interacts with the web application on the client-side, an HTTP request is sent to the server. This request goes through the Application Layer, Transport Layer (TCP), Network Layer (IP), Data Link Layer, and finally, the Physical Layer, where the raw bits are transmitted over the network.

The server receives the request and processes it using Flask. If the request is a valid fan speed command, the server updates the fan speed and emits a WebSocket event ('fan_speed_update') to all connected clients using Flask-SocketIO. This event is sent back to the clients through the same layers, but this time using the WebSocket protocol instead of HTTP.

The client receives the WebSocket event and updates the fan animation accordingly, providing a real-time update of the fan speed without the need for a full page refresh.