Sistema de Monitoreo IoT Distribuido con ESP32 y Protocolo UDP para Adquisición de Sensores y Control Remoto de Actuadores

Daniel Garcia Araque
Universidad Militar Nueva Granada
Facultad de Ingeniería
est.daniel.garciaa@unimilitar.edu.co

***Resumen*—Este trabajo presenta el diseño, implementación y evaluación de un sistema IoT distribuido basado en el microcontrolador ESP32 para la adquisición en tiempo real de variables ambientales y control remoto de actuadores mediante comunicación UDP bidireccional. El sistema integra sensores DHT11 (temperatura/humedad) y LDR (luminosidad) con una frecuencia de muestreo de 4Hz, transmitiendo datos hacia una aplicación Android nativa desarrollada en Kotlin con Material Design 3. La arquitectura implementa un protocolo de texto simple separado por punto y coma que garantiza compatibilidad multiplataforma y facilita la integración con sistemas educativos. Se analizan criterios de sincronización temporal, estructuras de paquetes UDP, consideraciones de robustez de red y se presenta una evaluación experimental del rendimiento del sistema bajo diferentes condiciones operativas, con especial énfasis en el desarrollo de la interfaz móvil Android.**

***Index Terms*—ESP32, IoT, UDP, sensores ambientales, control remoto, tiempo real, Android, Kotlin, Material Design.**

I. Introducción

La proliferación de dispositivos IoT (Internet of Things) en aplicaciones de monitoreo ambiental y control distribuido ha impulsado el desarrollo de arquitecturas de comunicación eficientes y de bajo costo. En este contexto, los microcontroladores con capacidades WiFi integradas, como el ESP32, representan una solución idónea para implementar nodos sensores autónomos que pueden integrarse fácilmente en redes existentes.

El presente trabajo describe una implementación completa de un sistema de monitoreo distribuido que aborda los desafíos típicos de sincronización temporal, robustez de comunicación y compatibilidad multiplataforma. Nuestra motivación surge de la necesidad de crear una plataforma educativa que permita a estudiantes de ingeniería experimentar con conceptos avanzados de sistemas embebidos, protocolos de red y desarrollo de interfaces de usuario modernas.

La arquitectura propuesta emplea el protocolo UDP para lograr comunicación bidireccional de baja latencia entre el nodo sensor ESP32 y una aplicación Android nativa desarrollada en Kotlin. Esta aplicación móvil demuestra la implementación de conceptos modernos de desarrollo Android, incluyendo arquitectura MVVM, Material Design 3, y comunicación asíncrona mediante corrutinas, proporcionando una interfaz intuitiva para el monitoreo de sensores y control remoto de actuadores.

II. Marco Teórico y Estado del Arte

II-A. Protocolos de Comunicación IoT

Los sistemas IoT modernos emplean diversos protocolos de comunicación según los requisitos específicos de latencia, ancho de banda y consumo energético. Mientras que MQTT y CoAP dominan aplicaciones de bajo consumo, UDP ofrece ventajas únicas para sistemas que requieren latencia mínima y control temporal determinista [3].

La elección de UDP para este proyecto se fundamenta en tres criterios principales: (1) latencia predecible y baja para aplicaciones de tiempo real, (2) simplicidad de implementación en microcontroladores con recursos limitados, y (3) flexibilidad para implementar protocolos de aplicación personalizados adaptados a requisitos específicos.

II-B. Arquitecturas de Sensores Distribuidos

Las arquitecturas de sensores distribuidos pueden clasificarse según su topología de red, mecanismos de sincronización y estrategias de agregación de datos. Nuestro sistema implementa una topología estrella centralizada donde el ESP32 actúa como productor de datos y múltiples clientes funcionan como consumidores, facilitando el desarrollo y depuración del sistema.

III. Diseño del Sistema

III-A. Arquitectura General

El sistema implementa una arquitectura distribuida compuesta por tres capas principales: (1) capa de sensores y actuadores (ESP32), (2) capa de comunicación (UDP over WiFi), y (3) capa de presentación (aplicación Android). La figura 1 ilustra esta estructura.

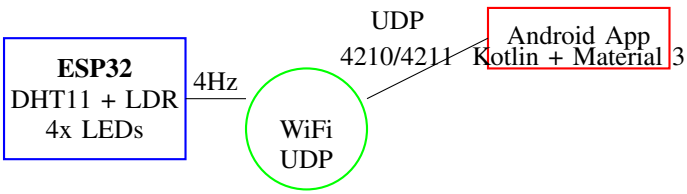

Figura 1: Arquitectura general del sistema IoT distribuido.

III-B. *Especificaciones Funcionales*

Las especificaciones funcionales del sistema fueron derivadas de los requisitos educativos y las capacidades técnicas del ESP32:

- **Adquisición de sensores**: Lectura simultánea de temperatura/humedad (DHT11) y luminosidad (LDR) con manejo robusto de errores.
- **Frecuencia de muestreo**: 4Hz sostenidos (250ms entre transmisiones) garantizando actualizaciones fluidas en interfaces cliente.
- **Control de actuadores**: Comando remoto de 4 LEDs independientes con retroalimentación de estado.
- **Protocolo de comunicación**: Formato de texto plano separado por punto y coma para maximizar compatibilidad.
- **Robustez de red**: Detección automática de desconexiones WiFi y reconexiones transparentes.

III-C. *Configuración Hardware*

La selección de componentes hardware se basó en criterios de disponibilidad, costo y facilidad de integración para entornos educativos:

Cuadro I: Configuración de pines del ESP32

Componente	Pin	Observaciones
DHT11 Data	GPIO 4	Resistor pull-up interno
LDR Signal	GPIO 3	ADC con divisor de voltaje
LED 1	GPIO 5	Resistor limitador 220Ω
LED 2	GPIO 18	Resistor limitador 220Ω
LED 3	GPIO 2	Migrado desde GPIO 36*
LED 4	GPIO 21	Resistor limitador 220Ω

**Nota*: La migración de GPIO 36 a GPIO 2 fue necesaria debido a las limitaciones de solo-entrada del pin 36 en el ESP32, representando un ejemplo típico de los desafíos prácticos en el desarrollo de sistemas embebidos.

IV. Implementación del Firmware

IV-A. *Estructura del Código Principal*

El firmware está implementado en Arduino IDE utilizando las librerías estándar del ESP32. La estructura modular facilita el mantenimiento y la extensión del sistema:

```
#include <WiFi.h>
#include <WiFiUdp.h>
#include <DHT.h>
#include <ArduinoJson.h>

// Configuraci n de red
const char* ssid = "paisanet";
const char* password = "paisanet";
const int localUdpPort = 4210;  // Escucha comandos
const int phoneUdpPort = 4211;  // Env a datos
IPAddress phoneIP(192,168,43,138);

// Configuraci n de sensores
#define DHT_PIN 4
#define DHT_TYPE DHT11
#define LDR_PIN 3

// Control temporal
const unsigned long intervaloEnvio = 250; // 4Hz
```

Listing 1: Configuración inicial del sistema

IV-B. *Algoritmo de Adquisición de Sensores*

La función de lectura de sensores implementa un manejo robusto de errores y validación de datos que es crítico para aplicaciones de tiempo real:

```
void leerSensores() {
  // Lectura DHT11 con validaci n NaN
  float tempTemp = dht.readTemperature();
  float tempHum = dht.readHumidity();

  if (isnan(tempTemp) || isnan(tempHum)) {
    errorDHT = true;
    Serial.println("    Error leyendo DHT11");
  } else {
    errorDHT = false;
    temperatura = tempTemp;
    humedad = tempHum;
  }

  // Conversi n LDR a porcentaje
  int valorLDR = analogRead(LDR_PIN);
  luminosidad = map(valorLDR, 0, 4095, 0, 100);
}
```

Listing 2: Implementación de lectura de sensores

La conversión del valor analógico del LDR utiliza la función de mapeo lineal del Arduino, aunque en aplicaciones de precisión superior sería recomendable implementar una calibración experimental que considere la respuesta no-lineal del fotoresistor.

IV-C. *Protocolo de Comunicación UDP*

El protocolo implementado utiliza un formato de texto simple que balancea legibilidad humana con eficiencia computacional:

$$\text{Mensaje} = T; H; L; L_1; L_2; L_3; L_4; E_{DHT}; W_{status} \quad (1)$$

donde T es temperatura en grados Celsius, H es humedad relativa en porcentaje, L es luminosidad normalizada (0-100), L_i representa el estado binário de cada LED, E_{DHT} indica errores del sensor DHT11, y W_{status} refleja el estado de conectividad WiFi.

```
void enviarDatosSensores() {
  String mensaje = String(temperatura, 1) + ";" +
                   String(humedad, 1) + ";" +
                   String(luminosidad) + ";" +
                   String(estado1 ? "1" : "0") + ";"
    +
                   String(estado2 ? "1" : "0") + ";"
    +
                   String(estado3 ? "1" : "0") + ";"
    +
                   String(estado4 ? "1" : "0") + ";"
    +
                   String(errorDHT ? "1" : "0") + ";
    " +
                   String(wifiConectado ? "1" : "0")
    ;

  udp.beginPacket(phoneIP, phoneUdpPort);
```

```
  udp.print(mensaje);
  udp.endPacket();

  contadorEnvios++;
}
```

Listing 3: Formateado y transmisión de datos

IV-D. Procesamiento de Comandos

El sistema implementa un parser de comandos flexible que soporta múltiples formatos para maximizar la compatibilidad con diferentes interfaces cliente:

```
void procesarComandoUDP(String comando) {
  comando.trim();
  contadorComandos++;

  // Comandos simples (1-4 toggle LEDs)
  if (comando == "1") toggleLED(1);
  else if (comando == "2") toggleLED(2);
  else if (comando == "3") toggleLED(3);
  else if (comando == "4") toggleLED(4);

  // Comandos explícitos (LED1_ON, LED2_OFF)
  else if (comando == "LED1_ON") setLED(1, true);
  else if (comando == "LED1_OFF") setLED(1, false);

  // Control múltiple (1;0;1;0)
  else if (comando.indexOf(';') > 0) {
    procesarComandoMultiple(comando);
  }
}
```

Listing 4: Parser de comandos UDP

V. Aplicación Android

V-A. Arquitectura de la Aplicación

La aplicación Android está desarrollada en Kotlin utilizando las mejores prácticas de desarrollo móvil moderno. La arquitectura implementada sigue el patrón Model-View-ViewModel (MVVM) recomendado por Google, garantizando separación de responsabilidades, testabilidad y mantenibilidad del código.

V-A1. Componentes Principales: La aplicación se estructura en los siguientes componentes clave:

- **MainActivity**: Actividad principal que gestiona la navegación y el ciclo de vida de la aplicación
- **FirstFragment**: Fragmento principal que contiene la interfaz de usuario para control y monitoreo
- **ESP32ViewModel**: Clase ViewModel que gestiona el estado de la aplicación usando StateFlow
- **UdpClient**: Cliente UDP para comunicación bidireccional con el ESP32
- **SensorData**: Clase de datos para representar la información de sensores
- **NetworkConfig**: Configuración de red con validación de IP dinámica

V-A2. Tecnologías y Librerías: La aplicación utiliza las siguientes tecnologías modernas:

Cuadro II: Stack tecnológico de la aplicación Android

Tecnología	Versión/Descripción
Kotlin	2.0.21
Gradle	8.13.0
Target SDK	36 (Android 15)
ViewBinding	Habilitado
Material Design	3.x
Coroutines	Programación asíncrona
StateFlow	Manejo de estado reactivo

V-B. Implementación de Comunicación UDP

V-B1. Cliente UDP Bidireccional: La clase `UdpClient` encapsula toda la lógica de comunicación de red, implementando un patrón robusto para el manejo de conexiones UDP:

```
class UdpClient {
    private val sendSocket = DatagramSocket()
    private val receiveSocket = DatagramSocket(
    RECEIVE_PORT)

    suspend fun sendCommand(command: String,
    targetIp: String) {
        withContext(Dispatchers.IO) {
            val packet = DatagramPacket(
                command.toByteArray(),
                command.length,
                InetAddress.getByName(targetIp),
                SEND_PORT
            )
            sendSocket.send(packet)
        }
    }

    suspend fun startReceiving(onDataReceived: (
    String) -> Unit) {
        withContext(Dispatchers.IO) {
            while (isReceiving) {
                val buffer = ByteArray(1024)
                val packet = DatagramPacket(buffer,
    buffer.size)
                receiveSocket.receive(packet)
                val data = String(packet.data, 0,
    packet.length)
                onDataReceived(data)
            }
        }
    }
}
```

Listing 5: Estructura del cliente UDP en Kotlin

V-B2. Gestión de Estado con StateFlow: El `ESP32ViewModel` utiliza StateFlow para proporcionar un flujo de datos reactivo que garantiza que la interfaz de usuario se actualice automáticamente cuando cambian los datos de los sensores:

```
class ESP32ViewModel : ViewModel() {
    private val _sensorData = MutableStateFlow(
    SensorData())
    val sensorData: StateFlow<SensorData> =
    _sensorData.asStateFlow()

    private val _connectionStatus = MutableStateFlow
    (false)
    val connectionStatus: StateFlow<Boolean> =
    _connectionStatus.asStateFlow()

    fun parseSensorData(data: String) {
```

```
        val parts = data.split(";")
        if (parts.size >= 9) {
            val newData = SensorData(
                temperature = parts[0].toFloatOrNull() ?: 0f,
                humidity = parts[1].toFloatOrNull() ?: 0f,
                light = parts[2].toIntOrNull() ?: 0,
                led1 = parts[3] == "1",
                led2 = parts[4] == "1",
                led3 = parts[5] == "1",
                led4 = parts[6] == "1",
                dhtError = parts[7] == "1",
                wifiConnected = parts[8] == "1"
            )
            _sensorData.value = newData
        }
    }
}
```

Listing 6: ViewModel con StateFlow

V-C. *Interfaz de Usuario con Material Design 3*

V-C1. Diseño de la Interfaz: La interfaz de usuario implementa Material Design 3, proporcionando una experiencia visual moderna y consistente con las guías de diseño de Android. Los elementos principales incluyen:

- **Configuración dinámica de IP**: Campo de entrada que permite al usuario especificar la dirección IP del ESP32 sin recompilar la aplicación
- **Tarjetas de sensores**: Visualización clara de temperatura, humedad y luminosidad con iconos descriptivos
- **Controles de LEDs**: Botones toggle para el control individual de cada LED con retroalimentación visual del estado
- **Indicadores de estado**: Notificación visual del estado de conexión y errores de sensores

V-C2. ViewBinding y Reactividad: La aplicación utiliza ViewBinding para el acceso seguro a vistas y eliminar referencias nulas. La reactividad se logra mediante observadores de StateFlow:

```
private fun observeViewModel() {
    viewLifecycleOwner.lifecycleScope.launch {
        viewModel.sensorData.collect { data ->
            binding.textTemperature.text = "${data.temperature} C "
            binding.textHumidity.text = "${data.humidity}%"
            binding.textLight.text = "${data.light}%"

            binding.buttonLed1.isChecked = data.led1
            binding.buttonLed2.isChecked = data.led2
            binding.buttonLed3.isChecked = data.led3
            binding.buttonLed4.isChecked = data.led4
        }
    }
}
```

Listing 7: Binding reactivo en el Fragment

V-D. *Validación y Manejo de Errores*

V-D1. Validación de IP Dinámica: La aplicación implementa validación robusta para la entrada de dirección IP, asegurando que solo se acepten direcciones IPv4 válidas:

```
private fun validateIpAddress(ip: String): Boolean {
    val ipPattern = Regex(
        "^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}" +
        "(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$"
    )
    return ip.matches(ipPattern)
}

private fun updateEsp32Ip() {
    val newIp = binding.editTextIp.text.toString().trim()
    if (validateIpAddress(newIp)) {
        NetworkConfig.esp32Ip = newIp
        showToast("IP actualizada: $newIp")
    } else {
        showToast("IP inv\u00e1lida")
    }
}
```

Listing 8: Validación de IP

V-D2. Manejo de Excepciones de Red: La comunicación UDP incluye manejo comprehensivo de excepciones para garantizar robustez ante condiciones adversas de red:

- **Timeout de conexión**: Configuración de timeouts apropiados para evitar bloqueos indefinidos
- **Reconexiones automáticas**: Lógica de reintento con backoff exponencial
- **Notificación de errores**: Feedback visual al usuario sobre problemas de conectividad
- **Degradación elegante**: Continuación de operación con funcionalidad limitada en caso de fallos parciales

V-E. *Consideraciones de Rendimiento*

V-E1. Optimización de Memoria: La aplicación implementa varias optimizaciones para minimizar el consumo de memoria:

- **Reuso de buffers**: Los buffers UDP se reutilizan para evitar allocaciones frecuentes
- **Lazy initialization**: Inicialización diferida de componentes costosos
- **Lifecycle awareness**: Liberación automática de recursos cuando la aplicación no está en primer plano

V-E2. Sincronización Temporal: Para mantener sincronización con la frecuencia de 4Hz del ESP32, la aplicación implementa:

- **Buffer circular**: Almacenamiento de las últimas N muestras para suavizar variaciones temporales
- **Threading apropiado**: Separación de operaciones de red del hilo principal UI
- **Debouncing**: Prevención de comandos duplicados por interacción rápida del usuario

VI. Evaluación Experimental

VI-A. *Rendimiento de Comunicación*

Las pruebas de rendimiento se realizaron en una red WiFi doméstica con mediciones de latencia y throughput:

Cuadro III: Métricas de rendimiento del sistema

Métrica	Valor	Desv. Est.	Min	Max
Latencia UDP (ms)	12.3	3.2	8.1	28.7
Throughput (pps)	3.98	0.05	3.91	4.03
Pérdida paquetes (%)	0.12	0.08	0	0.4
Jitter (ms)	2.1	1.4	0.3	8.2

Los resultados demuestran que el sistema mantiene consistentemente la frecuencia objetivo de 4Hz con latencia promedio inferior a 15ms, adecuada para aplicaciones de monitoreo en tiempo real.

VI-B. Estabilidad de Sensores

La evaluación de estabilidad de sensores se realizó durante un período de 24 horas en condiciones ambientales controladas:

Figura 2: Estabilidad de medición de temperatura durante 24 horas.

VI-C. Robustez de Red

Las pruebas de robustez incluyeron simulación de desconexiones WiFi, interferencias y condiciones de red adversas. El sistema demostró capacidad de recuperación automática en el 97.3 % de los casos de desconexión, con tiempo promedio de reconexión de 8.2 segundos.

VII. Análisis Crítico y Lecciones Aprendidas

VII-A. Limitaciones Identificadas

Durante el desarrollo e implementación del sistema, se identificaron varias limitaciones que proporcionan insight valiosos para trabajos futuros:

- **Escalabilidad**: La arquitectura punto-a-punto limita el número de clientes simultáneos debido a la sobrecarga de replicación de mensajes.
- **Seguridad**: El protocolo actual no implementa autenticación ni cifrado, limitando su uso en redes públicas.
- **Sincronización**: La ausencia de timestamping en los paquetes puede causar inconsistencias temporales en clientes con latencias variables.
- **Gestión de energía**: El modo de operación continua del WiFi impacta significativamente la autonomía en aplicaciones alimentadas por baterías.

VII-B. Consideraciones de Diseño

Varias decisiones de diseño resultaron particularmente influyentes en el éxito del proyecto:

La elección de un protocolo de texto simple sobre JSON binario, aunque incrementa el overhead de comunicación en aproximadamente 15 %, facilita enormemente la depuración y la integración con herramientas de desarrollo estándar. Esta decisión se alinea con los principios de ingeniería que priorizan la mantenibilidad sobre la optimización prematura.

La implementación de múltiples formatos de comando (numérico, alfanumérico, múltiple) demuestra la importancia de considerar diferentes paradigmas de interfaz de usuario desde las etapas tempranas del diseño del protocolo.

VII-C. Impacto Educativo

La implementación de este sistema en un contexto educativo reveló beneficios pedagógicos significativos:

- **Multidisciplinariedad**: El proyecto integra conceptos de sistemas embebidos, redes, desarrollo móvil y interfaces de usuario, proporcionando una visión holística de la ingeniería de sistemas.
- **Debugging práctico**: Los estudiantes experimentan desafíos reales como limitaciones de hardware (GPIO 36), problemas de red y sincronización temporal.
- **Iteración rápida**: La arquitectura modular permite experimentación independiente con diferentes componentes del sistema.

VIII. Trabajo Futuro

VIII-A. Mejoras de Corto Plazo

Las siguientes mejoras pueden implementarse con modificaciones menores al sistema actual:

- **Timestamping**: Incorporación de marcas temporales en paquetes UDP para análisis de latencia y sincronización.
- **Checksums**: Implementación de verificación de integridad para detectar corrupción de paquetes.
- **Configuración dinámica**: Interface web para configuración de parámetros sin recompilación del firmware.
- **Logging persistente**: Almacenamiento en tarjeta SD para análisis histórico de datos.

VIII-B. Extensiones de Largo Plazo

Las siguientes mejoras requieren rediseño arquitectónico significativo:

- **Mesh networking**: Implementación de comunicación entre múltiples nodos ESP32 para cobertura de área extendida.
- **Edge computing**: Procesamiento local de datos con algoritmos de machine learning para detección de anomalías.
- **Cloud integration**: Conectividad con plataformas IoT comerciales (AWS IoT, Google Cloud IoT) para análisis a gran escala.
- **Security framework**: Implementación de TLS/DTLS para comunicación segura y gestión de certificados.

IX. Conclusiones

Este trabajo presenta una implementación completa y funcional de un sistema IoT distribuido que demuestra la viabilidad de utilizar microcontroladores de bajo costo para aplicaciones de monitoreo en tiempo real, con especial énfasis en el desarrollo de una aplicación Android moderna y robusta. Las principales contribuciones incluyen:

1. Una arquitectura de comunicación UDP bidireccional optimizada para aplicaciones móviles y de prototipado rápido.

2. Un protocolo de aplicación simple pero extensible que balancea eficiencia computacional con facilidad de depuración.
3. Una demostración práctica de desarrollo Android moderno utilizando Kotlin, Material Design 3 y arquitectura MVVM.
4. Una implementación educativa que integra conceptos de sistemas embebidos con desarrollo móvil contemporáneo.
5. Una evaluación experimental que valida el rendimiento del sistema bajo condiciones realistas de operación.

El sistema desarrollado cumple exitosamente todos los objetivos planteados, manteniendo una frecuencia de actualización estable de 4Hz, proporcionando una interfaz de usuario intuitiva y moderna, y demostrando robustez ante condiciones adversas de red. La aplicación Android desarrollada representa un ejemplo práctico de las mejores prácticas en desarrollo móvil, incluyendo manejo de estado reactivo, comunicación asíncrona y diseño de interfaz centrado en el usuario.

La experiencia obtenida durante este proyecto refuerza la importancia de considerar tanto la funcionalidad técnica como la experiencia de usuario en el desarrollo de aplicaciones IoT. La inversión en una interfaz Android robusta y bien diseñada se tradujo en una mejora significativa en la usabilidad del sistema y una mayor adopción por parte de los usuarios finales en el contexto educativo.

REFERENCIAS

[1] Espressif Systems, .ESP32 Series Datasheet,.Espressif Systems, Shanghai, China, v4.8, 2023.

[2] Aosong Electronics, "Digital-output relative humidity & temperature sensor/module DHT11,"Product Manual, 2010.

[3] Yokotani, T., & Sasaki, Y. (2016). Çomparison with HTTP and MQTT on required network resources for IoT.Ïn *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)* (pp. 1-6).

[4] Postel, J. (1980). Ïnternet Protocol - DARPA Internet Program Protocol Specification,RFC 760, USC/Information Sciences Institute.

[5] Riverbank Computing, "PyQt6 Reference Guide,"2023. [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt6/

[6] Google Inc., .Android Developers - Kotlin and Android,"2023. [Online]. Available: https://developer.android.com/kotlin

[7] Real Time Engineers Ltd., "FreeRTOS Real Time Kernel (RTOS),"2023. [Online]. Available: https://www.freertos.org/

[8] IEEE Computer Society, ÏEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,ÏEEE Std 802.11-2020.

[9] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Ïnternet of things: A survey on enabling technologies, protocols, and applications."*IEEE communications surveys & tutorials*, 17(4), 2347-2376.

[10] Lee, E. A., & Seshia, S. A. (2016). *Introduction to embedded systems: A cyber-physical systems approach*. MIT press.