Bluetooth Communication and Implementation in a Mobile Application

Prof. Claudio Enrico Palazzi
Department of Mathematics
University of Padua
Padua, Italy
cpalazzi@math.unipd.it

Marco Cappellari
Department of Mathematics
University of Padua
Padua, Italy
marco.cappellari.4@studenti.unipd.it

Andrea Giurisato
Department of Mathematics
University of Padua
Padua, Italy
andrea.giurisato@studenti.unipd.it

***Abstract*—This paper explores the implementation of Bluetooth communication in a mobile application. It provides an overview of Bluetooth technology, discusses scanning, pairing, and data transfer processes, and presents a practical implementation within an application. Finally, manual tests conducted on the system are described, concluding with an evaluation of usability.**

Bluetooth has revolutionized short-range wireless communication by enabling seamless connectivity between devices. This paper provides a comprehensive examination of Bluetooth technology, from discovery to data exchange, and demonstrates its implementation in a mobile-based application. The research highlights both theoretical and practical aspects, providing insights into real-world Bluetooth application development and testing. The paper also discusses potential challenges, optimizations, and future advancements in Bluetooth technology to improve its efficiency and security in modern applications.

***Index Terms*—Bluetooth, Wireless Communication, Bluetooth Classic, Device Scanning, Pairing, Connection, Data Transfer, Mobile Application, Android.**

I. Introduction

Bluetooth is a widely used wireless technology for short-range communication. Originally developed in 1994 by Ericsson, Bluetooth operates in the 2.4 GHz ISM band and is designed to enable wireless communication between electronic devices. The technology has undergone significant advancements, with each new version bringing improvements in speed, range, security, and power consumption.

Bluetooth is now an integral part of modern communication, enabling a wide range of applications from audio streaming and file sharing to home automation and industrial IoT systems. Its versatility stems from its ability to operate in both point-to-point and multipoint configurations, making it suitable for various use cases across different industries.

Beyond different versions, Bluetooth exists in various types based on the application:

A. Bluetooth Classic

- Used in audio devices (headphones, speakers), game controllers, computers, and phones.
- Bluetooth supports moderate data transfer rates and continuous data transmission, providing stable connections, although interruptions may occur depending on the environment and distance between devices.

B. Bluetooth Low Energy (BLE)

- Introduced with Bluetooth 4.0, BLE is designed for low-power applications such as wearable devices, smart home sensors, and health monitoring systems.
- Uses a different communication model compared to Bluetooth Classic, focusing on short bursts of data transmission rather than continuous streaming.
- Does not natively support classic audio streaming, but Bluetooth 5.2 introduced LE Audio, which enhances low-power audio transmission capabilities.

C. Bluetooth Mesh

- Introduced with Bluetooth 5.0, Bluetooth Mesh extends Bluetooth's capabilities by enabling device-to-device communication in a decentralized network.
- Used in large-scale deployments such as smart lighting systems, industrial automation, and building management systems.
- Supports many-to-many communication, allowing devices to relay messages across a network for extended coverage.

Bluetooth has seen numerous improvements in speed, security, and energy efficiency. Recent versions, such as Bluetooth 5.0 and 5.2, introduced significant enhancements in range and data transfer capabilities, increasing the technology's adaptability for both consumer and industrial applications.

The future of Bluetooth includes further progress in latency reduction, security features, and support for multiple simultaneously connected devices. Emerging trends indicate a greater emphasis on Bluetooth in IoT, automotive applications, and medical devices, with a focus on energy-efficient communication protocols and enhanced encryption standards to ensure data security.

By understanding the evolution of Bluetooth and its various implementations, developers and researchers can leverage its

capabilities to create innovative applications that enhance connectivity and user experience.

II. Scanning

Scanning is the process of discovering nearby Bluetooth devices, enabling connections and data exchange. In Android, scanning is managed through the `BluetoothAdapter` class, which allows the device to search for other Bluetooth-enabled devices in its proximity.

A scan can operate in two modes:

- **Inquiry Scan**: The device actively sends inquiry requests and listens for responses from nearby discoverable devices.
- **Page Scan**: The device listens for connection attempts from previously paired devices.

To start scanning in Android, the application must request the appropriate permissions and register a `BroadcastReceiver` to capture the discovered devices. The following code snippet demonstrates how to implement a device discovery receiver:

```
class FoundDeviceReceiver(private val onDeviceFound:
     (BluetoothDevice) -> Unit): BroadcastReceiver()
     {
    override fun onReceive(context: Context?, intent
        : Intent?) {
        if (intent?.action == BluetoothDevice.
            ACTION_FOUND) {
            val device = intent.getParcelableExtra(
                BluetoothDevice.EXTRA_DEVICE) as?
                BluetoothDevice
            device?.let(onDeviceFound)
        }
    }
}
```

Listing 1. Bluetooth Scanning Receiver

The scanning process is initiated using:

```
bluetoothAdapter?.startDiscovery()
```

Listing 2. Start Bluetooth Scanning

During scanning, the application continuously listens for `ACTION_FOUND` broadcasts, which contain information about detected devices. The scan results include:

- Device name (if available)
- MAC address
- Device class (e.g., audio device, smartphone)
- Signal strength (RSSI)

The scanning implementation in this project is handled by the `AndroidBluetoothController` class. This class ensures that discovered devices are captured and stored efficiently using a `MutableStateFlow` to provide reactive updates to the UI. Below is a snippet demonstrating how discovered devices are stored:

```
private val _scannedDevices = MutableStateFlow<List<
    BluetoothDeviceDomain>>(emptyList())
override val scannedDevices: StateFlow<List<
    BluetoothDeviceDomain>>
    get() = _scannedDevices.asStateFlow()
```

```
private val foundDeviceReceiver =
    FoundDeviceReceiver { device ->
    _scannedDevices.update { devices ->
        val newDevice = device.
            toBluetoothDeviceDomain()
        if (newDevice !in devices) devices +
            newDevice else devices
    }
}
```

Listing 3. Storing Discovered Devices

This ensures that duplicate devices are filtered out and only new devices are added to the list, preventing redundant entries in the user interface.

To enhance the efficiency of scanning, the system employs a dynamic scan approach. The scan process is initiated when the user requests a discovery session, and is automatically stopped after a predefined timeout to reduce resource consumption:

```
override fun startDiscovery() {
    if (!hasPermission(Manifest.permission.
        BLUETOOTH_SCAN)) return
    context.registerReceiver(foundDeviceReceiver,
        IntentFilter(BluetoothDevice.ACTION_FOUND))
    bluetoothAdapter?.startDiscovery()
}

override fun stopDiscovery() {
    if (!hasPermission(Manifest.permission.
        BLUETOOTH_SCAN)) return
    bluetoothAdapter?.cancelDiscovery()
}
```

Listing 4. Managing Scan Lifecycle

The use of permissions is also crucial. Since Android 12 (API level 31), applications require the `BLUETOOTH_SCAN` and `BLUETOOTH_CONNECT` permissions. These are checked before initiating any scanning or connection operations.

This project does not implement Bluetooth Low Energy (BLE) scanning. Instead, it relies solely on Bluetooth Classic for device discovery.

In summary, Bluetooth scanning is a crucial part of device discovery. This implementation optimizes efficiency by using state management, filtering, and proper lifecycle management while adhering to Android's evolving permission requirements.

III. Pairing and Connection

Once a device is discovered, the next step is to establish a connection through a pairing process. Pairing is required to authenticate devices and establish a trusted link for secure communication. In Android, pairing and connection management is handled using the `BluetoothDevice` class.

Pairing typically follows three modes:

- **Just Works**: No authentication required; devices automatically pair without user intervention.
- **PIN Entry**: Requires the user to manually enter a PIN code on both devices.
- **Numeric Comparison**: Displays a numeric code on both devices that the user must confirm.

The following implementation ensures that the application can monitor connection changes using a broadcast receiver:

```
class BluetoothStateReceiver(
    private val onStateChanged: (Boolean,
        BluetoothDevice) -> Unit
) : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent
        : Intent?) {
        val device = intent?.getParcelableExtra<
            BluetoothDevice>(BluetoothDevice.
            EXTRA_DEVICE)
        when(intent?.action) {
            BluetoothDevice.ACTION_ACL_CONNECTED ->
                onStateChanged(true, device ?:
                return)
            BluetoothDevice.ACTION_ACL_DISCONNECTED
                -> onStateChanged(false, device ?:
                return)
        }
    }
}
```

Listing 5. Bluetooth State Receiver

Pairing and connection logic is managed within the `AndroidBluetoothController` class. Below is a snippet demonstrating how devices are connected:

```
override fun connectToDevice(device:
    BluetoothDeviceDomain): Flow<ConnectionResult> {
    return flow {
        if (!hasPermission(Manifest.permission.
            BLUETOOTH_CONNECT)) {
            throw SecurityException("No
                BLUETOOTH_CONNECT permission")
        }

        currentClientSocket = bluetoothAdapter
            ?.getRemoteDevice(device.address)
            ?.createRfcommSocketToServiceRecord(UUID
                .fromString(SERVICE_UUID))
        stopDiscovery()

        currentClientSocket?.let { socket ->
            try {
                socket.connect()
                emit(ConnectionResult.
                    ConnectionEstablished)
                BluetoothDataTransferService(socket)
                    .also {
                    dataTransferService = it
                    emitAll(
                        it.listenForIncomingMessages
                            ()
                            .map { ConnectionResult.
                                TransferSucceeded(it
                                ) }
                    )
                }
            } catch(e: IOException) {
                socket.close()
                currentClientSocket = null
                emit(ConnectionResult.Error("
                    Connection was interrupted"))
            }
        }
    }.onCompletion {
        closeConnection()
    }.flowOn(Dispatchers.IO)
}
```

Listing 6. Connecting to Bluetooth Device

This implementation ensures that only paired devices can be connected and uses a secure RFCOMM socket for communication. Additionally, error handling is integrated to close the connection in case of failures.

To monitor and update connection status, a `StateFlow` is used:

```
private val _isConnected = MutableStateFlow(false)
override val isConnected: StateFlow<Boolean>
    get() = _isConnected.asStateFlow()
```

Listing 7. Connection State Management

This state is updated dynamically in response to connection events captured by `BluetoothStateReceiver`, ensuring real-time UI updates.

In summary, pairing and connection management are critical steps in Bluetooth communication. The implementation in this project follows best practices by ensuring secure device connections, handling errors gracefully, and updating the application state in real time to provide a seamless user experience.

IV. Data Transfer

Once a connection is established, devices can exchange data through input and output streams over a `BluetoothSocket`. Data transmission in this project is managed by the `BluetoothDataTransferService` class, which handles message sending and reception efficiently.

Data transfer relies on a standard approach using `InputStream` and `OutputStream`. The following implementation sends messages from one device to another:

```
class BluetoothDataTransferService(private val
    socket: BluetoothSocket) {
    suspend fun sendMessage(bytes: ByteArray):
        Boolean {
        return withContext(Dispatchers.IO) {
            try {
                socket.outputStream.write(bytes)
                true
            } catch (e: IOException) {
                e.printStackTrace()
                false
            }
        }
    }
}
```

Listing 8. Sending Bluetooth Data

Messages are read from the input stream in a continuous loop to ensure all incoming data is processed correctly:

```
fun listenForIncomingMessages(): Flow<
    BluetoothMessage> {
    return flow {
        if (!socket.isConnected) return@flow
        val buffer = ByteArray(1024)
        while (true) {
            val byteCount = try {
                socket.inputStream.read(buffer)
            } catch (e: IOException) {
                throw TransferFailedException()
            }
            emit(buffer.decodeToString(endIndex =
                byteCount).toBluetoothMessage(false)
                )
        }
    }.flowOn(Dispatchers.IO)
```

```
}
```

Listing 9. Listening for Incoming Messages

To structure and decode messages, a simple protocol is implemented using the `BluetoothMessageMapper` class. Each message is prefixed with the sender's name and delimited by a special character:

```
fun String.toBluetoothMessage(isFromLocalUser:
    Boolean): BluetoothMessage {
    val name = substringBeforeLast("#")
    val message = substringAfter("#")
    return BluetoothMessage(
        message = message,
        senderName = name,
        isFromLocalUser = isFromLocalUser
    )
}
```

Listing 10. Message Formatting

A common challenge in data transfer is message fragmentation due to buffer size limitations. However, this is not a concern in our case, as we only send 2-character coordinates, which are smaller than the buffer size. Therefore, we do not need to handle larger messages.

In summary, the data transfer implementation ensures reliable communication between devices by using well-structured message handling, real-time connection monitoring, and proper error management. This approach provides a seamless experience for the user while maintaining efficiency in Bluetooth communication.

V. The App

The developed application integrates Bluetooth communication using Jetpack Compose for UI management.

The main components of the application include:

A. Device Screen

The `DeviceScreen` is responsible for listing scanned and paired devices. It provides an interactive interface where users can initiate Bluetooth discovery and select devices for pairing.

```
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.
                SpaceAround
        ) {
            Button(onClick = onStartScan) {
                Text(text = "Start scan")
            }
            Button(onClick = onStopScan) {
                Text(text = "Stop scan")
            }
            Button(onClick = onStartServer) {
                Text(text = "Start server")
            }
        }
```

Listing 11. Device Screen Implementation caption

B. Tris Game Screen

The `TrisGameScreen` implements a tic-tac-toe game using Bluetooth communication. The game allows two users to play in real-time by exchanging moves over a Bluetooth connection.

The application handles incoming messages and updates the game state accordingly.

VI. Tests

To evaluate the performance of the implemented Bluetooth communication system, we conducted a series of manual tests. These tests focused on verifying the correct operation of scanning, pairing, connection establishment, and data transfer functionalities.

All tests were performed manually by the developers, without the use of automated testing tools. The evaluation process involved executing predefined test cases under controlled conditions, analyzing the system's behavior, and recording observed results.

The manual tests included the following:

- **Device Scanning**: Ensuring that all nearby Bluetooth devices were correctly discovered and displayed in the app.
- **Pairing and Connection**: Verifying that devices could be successfully paired and connected without unexpected failures.
- **Data Transfer**: Sending messages between paired devices and ensuring data integrity without noticeable latency.
- **User Experience Evaluation**: Testing the responsiveness of the application UI during Bluetooth operations.

Some challenges were identified during testing, such as occasional connection drops and minor delays in data transmission. These issues were mitigated by implementing error-handling mechanisms and optimizing connection stability.

Since these tests were conducted manually, future work includes the implementation of automated testing frameworks to ensure consistent evaluation across different devices and environments.

VII. Conclusions

This paper presented an implementation of Bluetooth communication in a mobile application, covering essential aspects such as device discovery, pairing, connection management, and data transfer. The study highlighted the practical challenges involved in developing a Bluetooth-based system and provided solutions to optimize performance and reliability.

The manual tests conducted demonstrated the effectiveness of the implemented system, confirming its ability to establish and maintain stable Bluetooth connections. However, occasional connection drops and minor delays in data transmission were observed.

While the application performed well under normal conditions, several improvements can be considered for future work:

- Implementing **automated testing frameworks** to enhance the robustness of the Bluetooth communication system.
- Switching from classic Bluetooth to Bluetooth Low Energy (BLE) to reduce power consumption and extend battery life.

Bluetooth technology continues to evolve, with advancements in BLE efficiency, data transfer rates, and security protocols. Future enhancements to the application will leverage these improvements to provide a more reliable and scalable Bluetooth communication system.

REFERENCES

[1] Bluetooth SIG, "Bluetooth Technology Overview," [Online]. Available: https://www.bluetooth.com.

[2] Android Developers, "Bluetooth Overview," [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth.

[3] Bluetooth SIG, *Bluetooth Core Specification Version 5.4*, Mar. 2023. [Online]. Available: https://www.bluetooth.com/specifications/

[4] Android Developers, "Bluetooth overview," *Android Developer Documentation*, Accessed: Oct. 20, 2025. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth