The world of Android wireless communications without Internet

DroidKaigi 2022

@fushiroyama

About me

Fumihiko Shiroyama

Sr. Software Engineer at Microsoft

Father of two girls

PhD Student at JAIST

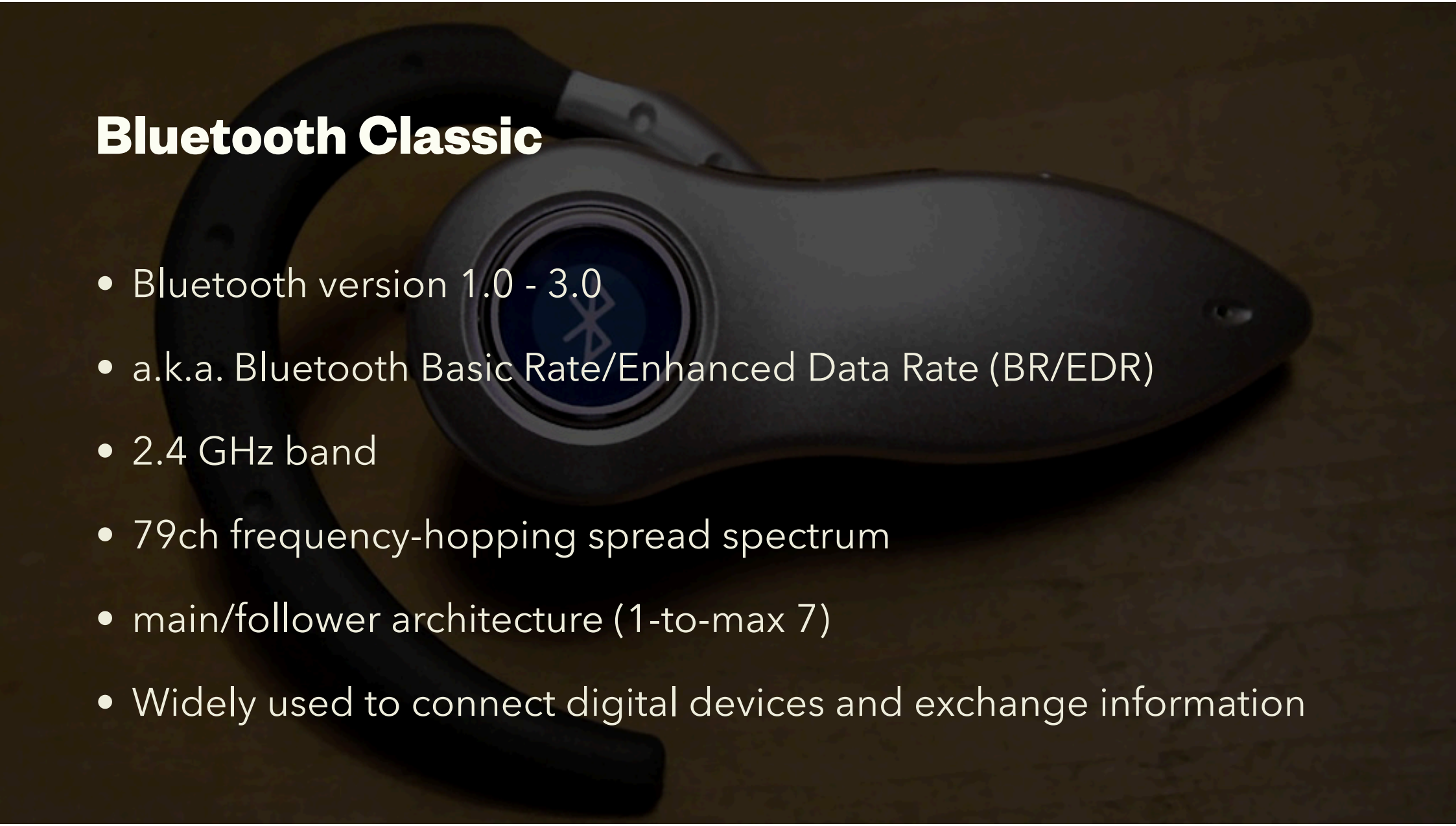
Agenda

- Overview of wireless communications without Internet
- Brief description of how to use them
- General guidelines on how to use each one differently
- Steps to more advanced topics

Overview of wireless communications without Internet

Wireless communications available for Android

- Bluetooth Classic
- Wi-Fi
- Bluetooth Low Energy
- NFC, etc..

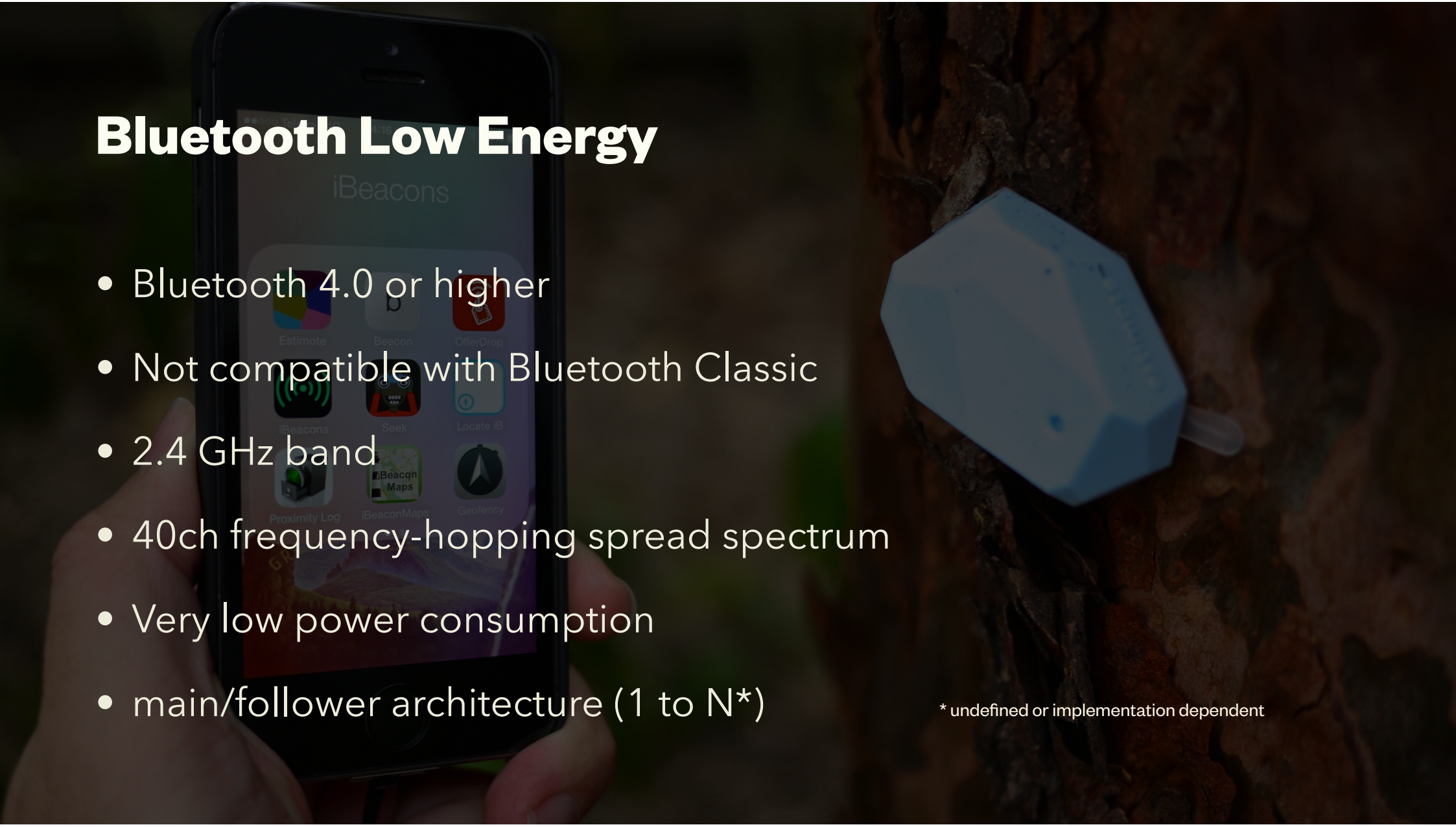


Bluetooth Classic

- Bluetooth version 1.0 - 3.0
- a.k.a. Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR)
- 2.4 GHz band
- 79ch frequency-hopping spread spectrum
- main/follower architecture (1-to-max 7)
- Widely used to connect digital devices and exchange information

Wi-Fi

- IEEE 802.11
- Wireless Ethernet
- Typically used in the star topology, but **Wi-Fi Direct** is available for P2P communication



Bluetooth Low Energy

- Bluetooth 4.0 or higher
- Not compatible with Bluetooth Classic
- 2.4 GHz band
- 40ch frequency-hopping spread spectrum
- Very low power consumption
- main/follower architecture (1 to N*)

* undefined or implementation dependent

NFC

- Near-field communication (NFC) is undoubtedly a form of short-range wireless communication, but it is typically ultra-short-range, less than 4 cm
- Not covered in this session

Brief description of how to use them

Sample code repository

- https://github.com/shiroyama/DroidKaigi2022


Wi-Fi Direct

Wi-Fi Direct

- Wi-Fi communication without an access point
- Can be handled exactly like a TCP socket once the connection is established

Permissions

```
<!-- WiFi Direct Permissions -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
```

Wi-Fi manager and channel

```
private lateinit var manager: WifiP2pManager
private lateinit var channel: WifiP2pManager.Channel

manager = getSystemService(Context.WIFI_P2P_SERVICE) as WifiP2pManager
channel = manager.initialize(applicationContext, mainLooper, null)
```

Intent filter for receiving Wi-Fi status

```
private val intentFilter: IntentFilter = object : IntentFilter() {
    init {
        addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION)
        addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION)
        addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION)
        addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION)
    }
}
```

BroadcastReceiver for Wi-Fi status

```
override fun onReceive(context: Context, intent: Intent) {
    when (intent.action) {
        WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION -> {}
    }
}
```

BroadcastReceiver for Wi-Fi status

```
override fun onReceive(context: Context, intent: Intent) {
    when (intent.action) {
        WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION -> {}
        WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION -> {}
    }
}
```

Register BroadcastReceiver

```
override fun onResume() {
    super.onResume()
    receiver = WiFiDirectBroadcastReceiver(manager, channel, this)
    registerReceiver(receiver, intentFilter)
}

override fun onPause() {
    unregisterReceiver(receiver)
    super.onPause()
}
```

Register BroadcastReceiver

```
override fun onResume() {
    super.onResume()
    receiver = WiFiDirectBroadcastReceiver(manager, channel, this)
    registerReceiver(receiver, intentFilter)
}

override fun onPause() {
    unregisterReceiver(receiver)
    super.onPause()
}
```

Discover peers

```
manager.discoverPeers(channel, object : WifiP2pManager.ActionListener {
    override fun onSuccess() {
        Log.d(TAG, "discoverPeers onSuccess")
    }

    override fun onFailure(i: Int) {
        Log.e(TAG, "discoverPeers onFailure: $i")
    }
})
```

When peer found

```
WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {
}
```

Request peer list

```
WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {
    manager.requestPeers(channel) { wifiP2pDeviceList: WifiP2pDeviceList ->
        val deviceList = wifiP2pDeviceList.deviceList
    }
}
```

Request peer list

```
WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {
    manager.requestPeers(channel) { wifiP2pDeviceList: WifiP2pDeviceList ->
        val deviceList = wifiP2pDeviceList.deviceList
    }
}
```

Request peer list

```
WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION -> {
    manager.requestPeers(channel) { wifiP2pDeviceList: WifiP2pDeviceList ->
        val deviceList = wifiP2pDeviceList.deviceList
    }
}
```

Connect to one of the peers

```
fun connect(device: WifiP2pDevice) {
    val config = WifiP2pConfig()
    config.deviceAddress = device.deviceAddress
    config.wps.setup = WpsInfo.PBC
    manager.connect(channel, config, object : WifiP2pManager.ActionListener {
        override fun onSuccess() {
            Log.d(TAG, "connect onSuccess")
        }

        override fun onFailure(i: Int) {
            Log.e(TAG, "connect onFailure: $i")
        }
    })
}
```

Connect to one of the peers

```
fun connect(device: WifiP2pDevice) {
    val config = WifiP2pConfig()
    config.deviceAddress = device.deviceAddress
    config.wps.setup = WpsInfo.PBC
    manager.connect(channel, config, object : WifiP2pManager.ActionListener {
        override fun onSuccess() {
            Log.d(TAG, "connect onSuccess")
        }

        override fun onFailure(i: Int) {
            Log.e(TAG, "connect onFailure: $i")
        }
    })
}
```

Connect to one of the peers

```
fun connect(device: WifiP2pDevice) {
    val config = WifiP2pConfig()
    config.deviceAddress = device.deviceAddress
    config.wps.setup = WpsInfo.PBC
    manager.connect(channel, config, object : WifiP2pManager.ActionListener {
        override fun onSuccess() {
            Log.d(TAG, "connect onSuccess")
        }

        override fun onFailure(i: Int) {
            Log.e(TAG, "connect onFailure: $i")
        }
    })
}
```

When connection status changes

```
WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
}
```

Get connection details

```
WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
    val networkInfo =
        intent.getParcelableExtra<NetworkInfo>(WifiP2pManager.EXTRA_NETWORK_INFO)
    val wifiP2pGroup =
        intent.getParcelableExtra<WifiP2pGroup>(WifiP2pManager.EXTRA_WIFI_P2P_GROUP)
    if (networkInfo?.isConnected == true) {
        manager.requestConnectionInfo(channel) { wifiP2pInfo: WifiP2pInfo ->
        }
    }
}
```

Get connection details

```
WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION -> {
    val networkInfo =
        intent.getParcelableExtra<NetworkInfo>(WifiP2pManager.EXTRA_NETWORK_INFO)
    val wifiP2pGroup =
        intent.getParcelableExtra<WifiP2pGroup>(WifiP2pManager.EXTRA_WIFI_P2P_GROUP)
    if (networkInfo?.isConnected == true) {
        manager.requestConnectionInfo(channel) { wifiP2pInfo: WifiP2pInfo ->
        }
    }
}
```

Connected

```
if (wifiP2pInfo.groupFormed && wifiP2pGroup?.isGroupOwner == true) {
    serverThread = ServerThread()
    serverThread.start()
} else if (wifiP2pInfo.groupFormed) {
    clientThread = ClientThread()
    clientThread.start()
} else {
    isConnected = false
}
```

Connected

```
if (wifiP2pInfo.groupFormed && wifiP2pGroup?.isGroupOwner == true) {
    serverThread = ServerThread()
    serverThread.start()
} else if (wifiP2pInfo.groupFormed) {
    clientThread = ClientThread()
    clientThread.start()
} else {
    isConnected = false
}
```

Connected

```
if (wifiP2pInfo.groupFormed && wifiP2pGroup?.isGroupOwner == true) {
    serverThread = ServerThread()
    serverThread.start()
} else if (wifiP2pInfo.groupFormed) {
    clientThread = ClientThread()
    clientThread.start()
} else {
    isConnected = false
}
```

Connected

```
if (wifiP2pInfo.groupFormed && wifiP2pGroup?.isGroupOwner == true) {
    serverThread = ServerThread()
    serverThread.start()
} else if (wifiP2pInfo.groupFormed) {
    clientThread = ClientThread()
    clientThread.start()
} else {
    isConnected = false
}
```

ServerThread

```
val serverSocket = ServerSocket(SOCKET_PORT)

// blocks until a connection is made
val socket: Socket = serverSocket.accept()
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ServerThread

```
val serverSocket = ServerSocket(SOCKET_PORT)

// blocks until a connection is made
val socket: Socket = serverSocket.accept()
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ServerThread

```
val serverSocket = ServerSocket(SOCKET_PORT)

// blocks until a connection is made
val socket: Socket = serverSocket.accept()
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ServerThread

```
val serverSocket = ServerSocket(SOCKET_PORT)

// blocks until a connection is made
val socket: Socket = serverSocket.accept()
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ClientThread

```
val socket = Socket()

// blocks until a connection is made
socket.connect(
    InetSocketAddress(wifiP2pInfo.groupOwnerAddress, SOCKET_PORT),
    SOCKET_TIMEOUT
)
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ClientThread

```
val socket = Socket()

// blocks until a connection is made
socket.connect(
    InetSocketAddress(wifiP2pInfo.groupOwnerAddress, SOCKET_PORT),
    SOCKET_TIMEOUT
)
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ClientThread

```
val socket = Socket()

// blocks until a connection is made
socket.connect(
    InetSocketAddress(wifiP2pInfo.groupOwnerAddress, SOCKET_PORT),
    SOCKET_TIMEOUT
)
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

ClientThread

```
val socket = Socket()

// blocks until a connection is made
socket.connect(
    InetSocketAddress(wifiP2pInfo.groupOwnerAddress, SOCKET_PORT),
    SOCKET_TIMEOUT
)
val inputStream: InputStream = socket.getInputStream()
val outputStream: OutputStream = socket.getOutputStream()
```

Wi-Fi Direct

- At this point, everything is just socket programming🔌
- Read message from the InputStream and write whatever you want to the OutputStream

Writing

```
val executorService = Executors.newSingleThreadExecutor()
executorService.submit {
    try {
        val inputStream = activity.resources.openRawResource(R.raw.sample)
        val buffer = ByteArray(8192)
        while (inputStream.read(buffer).also { length = it } != -1) {
            outputStream.write(buffer)
        }
    } catch (e: IOException) {
        Log.e(TAG, e.message, e)
    }
}
```

Writing

```
val executorService = Executors.newSingleThreadExecutor()
executorService.submit {
    try {
        val inputStream = activity.resources.openRawResource(R.raw.sample)
        val buffer = ByteArray(8192)
        while (inputStream.read(buffer).also { length = it } != -1) {
            outputStream.write(buffer)
        }
    } catch (e: IOException) {
        Log.e(TAG, e.message, e)
    }
}
```

Writing

```
val executorService = Executors.newSingleThreadExecutor()
executorService.submit {
    try {
        val inputStream = activity.resources.openRawResource(R.raw.sample)
        val buffer = ByteArray(8192)
        while (inputStream.read(buffer).also { length = it } != -1) {
            outputStream.write(buffer)
        }
    } catch (e: IOException) {
        Log.e(TAG, e.message, e)
    }
}
```

Reading

```
while (socket != null && !socket.isClosed) {
    // keep reading while connected
}
```

Reading

```
val buffer = ByteArray(8192)
try {
    val filename = System.currentTimeMillis().toString() + ".jpg"
    val fileOutputStream = context.openFileOutput(filename, Context.MODE_PRIVATE)
    var length: Int
    while (inputStream.read(buffer).also { length = it } != -1) {
        fileOutputStream.write(buffer, 0, length)
    }
    fileOutputStream.close()
} catch (e: IOException) {
    Log.e(TAG, e.message, e)
}
```

Reading

```
val buffer = ByteArray(8192)
try {
    val filename = System.currentTimeMillis().toString() + ".jpg"
    val fileOutputStream = context.openFileOutput(filename, Context.MODE_PRIVATE)
    var length: Int
    while (inputStream.read(buffer).also { length = it } != -1) {
        fileOutputStream.write(buffer, 0, length)
    }
    fileOutputStream.close()
} catch (e: IOException) {
    Log.e(TAG, e.message, e)
}
```

Reading

```
val buffer = ByteArray(8192)
try {
    val filename = System.currentTimeMillis().toString() + ".jpg"
    val fileOutputStream = context.openFileOutput(filename, Context.MODE_PRIVATE)
    var length: Int
    while (inputStream.read(buffer).also { length = it } != -1) {
        fileOutputStream.write(buffer, 0, length)
    }
    fileOutputStream.close()
} catch (e: IOException) {
    Log.e(TAG, e.message, e)
}
```

Tips

- **TCP has no such thing as a stream boundary**
- Messages may be sent and received in batches
- If you keep the socket open and continue the bidirectional communication, you need a protocol. For example, put the message length in the first few bytes.

Send messages with size

```
fun writeMessage(message: String) {
    val messageBytes = message.toByteArray(StandardCharsets.UTF_8)
    val length = messageBytes.size
    val lengthBytes = ByteBuffer.allocate(4).putInt(length).array()

    // writing the length of the message
    write(lengthBytes)

    // writing the message delimiter
    val delimiterBytes = ByteBuffer.allocate(2).putChar(':').array()
    write(delimiterBytes)

    // writing the message itself
    write(messageBytes)
}
```

Send messages with size

```
fun writeMessage(message: String) {
    val messageBytes = message.toByteArray(StandardCharsets.UTF_8)
    val length = messageBytes.size
    val lengthBytes = ByteBuffer.allocate(4).putInt(length).array()

    // writing the length of the message
    write(lengthBytes)

    // writing the message delimiter
    val delimiterBytes = ByteBuffer.allocate(2).putChar(':').array()
    write(delimiterBytes)

    // writing the message itself
    write(messageBytes)
}
```

Send messages with size

```
fun writeMessage(message: String) {
    val messageBytes = message.toByteArray(StandardCharsets.UTF_8)
    val length = messageBytes.size
    val lengthBytes = ByteBuffer.allocate(4).putInt(length).array()

    // writing the length of the message
    write(lengthBytes)

    // writing the message delimiter
    val delimiterBytes = ByteBuffer.allocate(2).putChar(':').array()
    write(delimiterBytes)

    // writing the message itself
    write(messageBytes)
}
```

Receive message with size

```
val lengthBuffer = ByteArray(4)
inputStream.read(lengthBuffer)
val length = ByteBuffer.wrap(lengthBuffer).int

val delimiterBuffer = ByteArray(2)
inputStream.read(delimiterBuffer)
val delimiter = ByteBuffer.wrap(delimiterBuffer).char
```

Receive message with size

```
val lengthBuffer = ByteArray(4)
inputStream.read(lengthBuffer)
val length = ByteBuffer.wrap(lengthBuffer).int

val delimiterBuffer = ByteArray(2)
inputStream.read(delimiterBuffer)
val delimiter = ByteBuffer.wrap(delimiterBuffer).char
```

Receive message with size

```
val pageSize = (length + BUFFER_SIZE - 1) / BUFFER_SIZE
var totalBytes = 0
val messageBuffer = ByteArray(length)
for (i in 0 until pageSize) {
    val limit = if (i == pageSize - 1) length - totalBytes else BUFFER_SIZE
    val incomingBytes = inputStream.read(INCOMING_BUFF, 0, limit)
    System.arraycopy(INCOMING_BUFF, 0, messageBuffer, totalBytes, incomingBytes)
    totalBytes += incomingBytes
    Log.d(TAG, "read(): totalBytes after reading = $totalBytes")
}
```

Receive message with size

```
val pageSize = (length + BUFFER_SIZE - 1) / BUFFER_SIZE
var totalBytes = 0
val messageBuffer = ByteArray(length)
for (i in 0 until pageSize) {
    val limit = if (i == pageSize - 1) length - totalBytes else BUFFER_SIZE
    val incomingBytes = inputStream.read(INCOMING_BUFF, 0, limit)
    System.arraycopy(INCOMING_BUFF, 0, messageBuffer, totalBytes, incomingBytes)
    totalBytes += incomingBytes
    Log.d(TAG, "read(): totalBytes after reading = $totalBytes")
}
```

Receive message with size

```
val pageSize = (length + BUFFER_SIZE - 1) / BUFFER_SIZE
var totalBytes = 0
val messageBuffer = ByteArray(length)
for (i in 0 until pageSize) {
    val limit = if (i == pageSize - 1) length - totalBytes else BUFFER_SIZE
    val incomingBytes = inputStream.read(INCOMING_BUFF, 0, limit)
    System.arraycopy(INCOMING_BUFF, 0, messageBuffer, totalBytes, incomingBytes)
    totalBytes += incomingBytes
    Log.d(TAG, "read(): totalBytes after reading = $totalBytes")
}
```

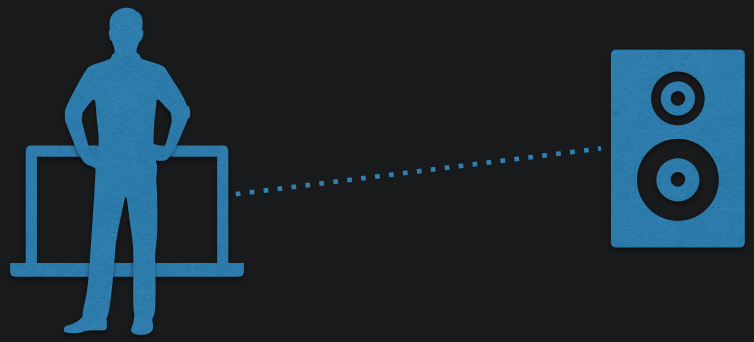
Wi-Fi Direct summary

- Wi-Fi Direct enables peer-to-peer communications without an access point on all devices that support Wi-Fi
- Once the connection is established, it is simple TCP socket communication. Both text and binary can be sent and received
- Pro's and con's to other wireless communications will be discussed later


Bluetooth Classic

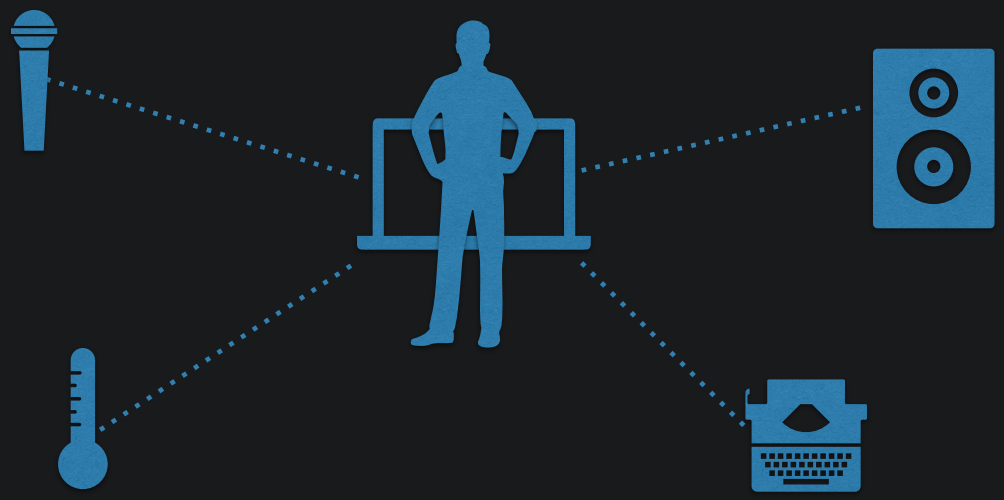
Bluetooth Classic

- main/follower (1 to 1)



Bluetooth Classic

- main/follower (1 to max 7)



Bluetooth Classic

- Profile: wireless interface specification for Bluetooth-based communication between device
  - Headset
  - Advanced Audio Distribution Profile (A2DP)
  - Health Device
  - https://developer.android.com/guide/topics/connectivity/bluetooth/profiles
- This talk will **NOT** cover those topics

Bluetooth Classic

- L2CAP (Logical Link Control and Adaptive Protocol)
  - In the OSI reference model, it is roughly equivalent to the data link layer (layer 2)
- RFCOMM (Radio Frequency Communication)
  - A protocol that emulates the RS-232C serial port on an L2CAP.
  - It has packet reception confirmation and retransmission like TCP.
  - In the OSI reference model, it is roughly equivalent to the transport layer (layer 4)
  - **RFCOMM sockets are the most common way to communicate in Android's Bluetooth API**

Permissions

```
<!-- Bluetooth Classic -->
<uses-permission
    android:name="android.permission.BLUETOOTH"
    android:maxSdkVersion="30" />
<uses-permission
    android:name="android.permission.BLUETOOTH_ADMIN"
    android:maxSdkVersion="30" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

<!-- Android 12 or higher -->
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
```

Permissions

```
<!-- Bluetooth Classic -->
<uses-permission
    android:name="android.permission.BLUETOOTH"
    android:maxSdkVersion="30" />
<uses-permission
    android:name="android.permission.BLUETOOTH_ADMIN"
    android:maxSdkVersion="30" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

<!-- Android 12 or higher -->
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
```

Check Bluetooth availability

```
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
if (bluetoothAdapter == null) {
    // Device doesn't support Bluetooth
}

if (bluetoothAdapter?.isEnabled == false) {
    val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT)
}
```

Check Bluetooth availability

```
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
if (bluetoothAdapter == null) {
    // Device doesn't support Bluetooth
}

if (bluetoothAdapter?.isEnabled == false) {
    val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT)
}
```

Check Bluetooth availability

```
val bluetoothAdapter: BluetoothAdapter? = BluetoothAdapter.getDefaultAdapter()
if (bluetoothAdapter == null) {
    // Device doesn't support Bluetooth
}

if (bluetoothAdapter?.isEnabled == false) {
    val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT)
}
```

BroadcastReceiver to get found device

```
private val receiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        when (intent.action) {
            BluetoothDevice.ACTION_FOUND -> {
                val device: BluetoothDevice? =
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
                device?.let {
                    val deviceName = it.name
                    val deviceAddress = it.address
                }
            }
        }
    }
}
```

BroadcastReceiver to get found device

```
private val receiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        when (intent.action) {
            BluetoothDevice.ACTION_FOUND -> {
                val device: BluetoothDevice? =
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
                device?.let {
                    val deviceName = it.name
                    val deviceAddress = it.address
                }
            }
        }
    }
}
```

Register BroadcastReceiver

```
override fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)

    val intentFilter = IntentFilter(BluetoothDevice.ACTION_FOUND)
    registerReceiver(receiver, intentFilter)
}

override fun onDestroy() {
    unregisterReceiver(receiver)
    super.onDestroy()
}
```

Make the device discoverable

```
val intent = Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE)
intent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, DISCOVERABLE_DURATION)
startActivityForResult(intent, REQUEST_CODE)
```

Make the device discoverable

```
val intent = Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE)
intent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, DISCOVERABLE_DURATION)
startActivityForResult(intent, REQUEST_CODE)
```

Start discovery

```
if (bluetoothAdapter.isDiscovering) {
    bluetoothAdapter.cancelDiscovery()
}
val discoveryResult = bluetoothAdapter.startDiscovery()
```

Start discovery

```
if (bluetoothAdapter.isDiscovering) {
    bluetoothAdapter.cancelDiscovery()
}
val discoveryResult = bluetoothAdapter.startDiscovery()
```

ServerThread

```
class BTServerThread : Thread() {
    private lateinit var bluetoothServerSocket: BluetoothServerSocket

    init {
        try {
            bluetoothServerSocket = bluetoothAdapter.listenUsingRfcommWithServiceRecord(
                BT_NAME, BT_UUID
            )
        } catch (e: IOException) {}
    }

    override fun run() {
        try {
            // blocks until connection
            val bluetooth: BluetoothSocket = bluetoothServerSocket.accept()
        } catch (e: IOException) {}
    }
```

ServerThread

```
class BTServerThread : Thread() {
    private lateinit var bluetoothServerSocket: BluetoothServerSocket

    init {
        try {
            bluetoothServerSocket = bluetoothAdapter.listenUsingRfcommWithServiceRecord(
                BT_NAME, BT_UUID
            )
        } catch (e: IOException) {}
    }

    override fun run() {
        try {
            // blocks until connection
            val bluetooth: BluetoothSocket = bluetoothServerSocket.accept()
        } catch (e: IOException) {}
    }
```

ServerThread

```
class BTServerThread : Thread() {
    private lateinit var bluetoothServerSocket: BluetoothServerSocket

    init {
        try {
            bluetoothServerSocket = bluetoothAdapter.listenUsingRfcommWithServiceRecord(
                BT_NAME, BT_UUID
            )
        } catch (e: IOException) {}
    }

    override fun run() {
        try {
            // blocks until connection
            val bluetooth: BluetoothSocket = bluetoothServerSocket.accept()
        } catch (e: IOException) {}
    }
```

Connect to server

```
fun connect(bluetoothDevice: BluetoothDevice) {
    btClientThread = BTClientThread(bluetoothDevice)
    btClientThread.start()
}
```

ClientThread

```
class BTClientThread(private val bluetoothDevice: BluetoothDevice) : Thread() {
    private lateinit var bluetoothSocket: BluetoothSocket

    init {
        try {
            bluetoothSocket = bluetoothDevice.createRfcommSocketToServiceRecord(BT_UUID)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
    }
```

ClientThread

```
class BTClientThread(private val bluetoothDevice: BluetoothDevice) : Thread() {
    private lateinit var bluetoothSocket: BluetoothSocket

    init {
        try {
            bluetoothSocket = bluetoothDevice.createRfcommSocketToServiceRecord(BT_UUID)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
    }
```

ClientThread

```
override fun run() {
    try {
        if (bluetoothAdapter.isDiscovering) {
            bluetoothAdapter.cancelDiscovery()
        }

        // blocks until connection
        bluetoothSocket.connect()
        ...
    } catch (e: IOException) {}
}
```

ClientThread

```
override fun run() {
    try {
        if (bluetoothAdapter.isDiscovering) {
            bluetoothAdapter.cancelDiscovery()
        }

        // blocks until connection
        bluetoothSocket.connect()
        ...
    } catch (e: IOException) {}
}
```

BluetoothSocket

```
val inputStream: InputStream = bluetoothSocket.inputStream
val outputStream: OutputStream = bluetoothSocket.outputStream
```

Same as TCP Socket!!

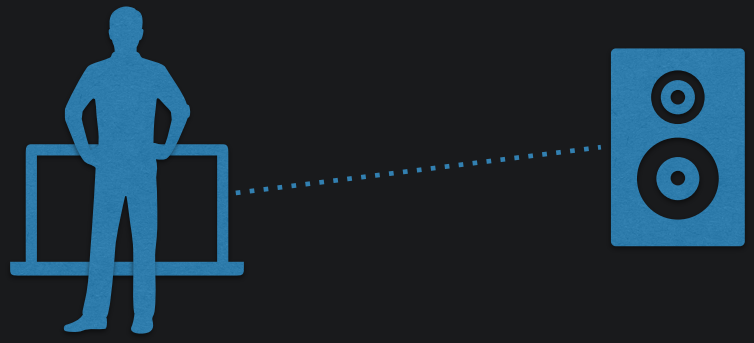
Bluetooth Classic summary

- **BluetoothSocket can be handled exactly like a TCP Socket**
- Thus, exactly the same applies when dealing with streams
- Pro's and con's to other wireless communications will be discussed later


Telstra 4G
4:16 pm
90%
iBeacons
Estimote
Beecon
OfferDrop
iBeacons
Proximity Log
iBeaconMaps
Geofency
Bluetooth Low Energy

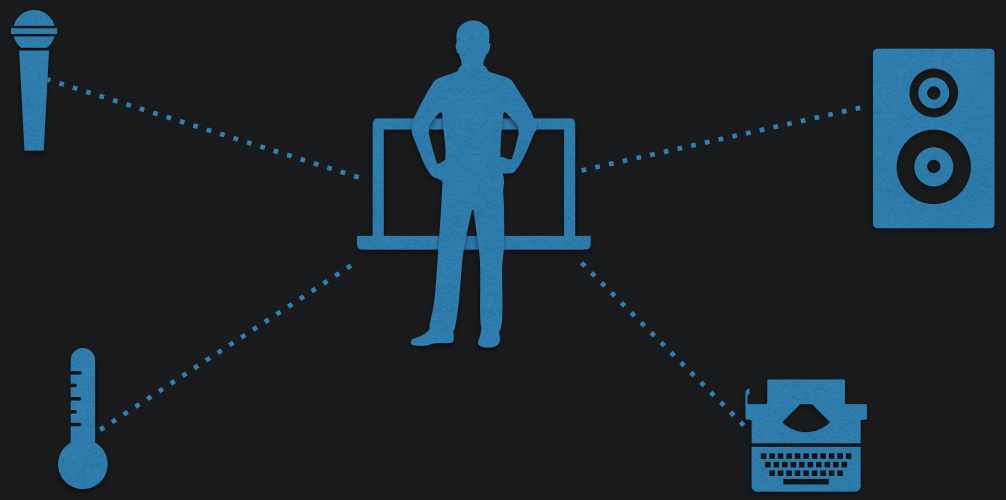
Bluetooth Low Energy

- a.k.a. Bluetooth LE or **BLE**
- main/follower (1 to 1)
- main is called **Central**, follower is called **Peripheral**



Bluetooth Low Energy

- central/peripheral (1 to N)



Bluetooth Low Energy

- Central and Peripheral are connected to form a main/follower relationship
- Peripheral can include a small amount of payload in advertised packets e.g.) iBeacon
- After connection, Central and Peripheral communicate using a method called **GATT** (Generic ATTribute profile)

GATT

- It defines the way in which Central reads and writes data to and from the Peripheral
- **Peripheral** is essentially a sensor device
- A Peripheral has one or more **Services**. This specifies what functions the peripheral provides
- A Service provides one or more **Characteristics**. This expresses the characteristics of the service.

GATT

Peripheral (thermometer)

Service (temperature information)

UUID: 5341560d-1e92-467c-8180-52e6c0a04d38

Characteristic (temperature)

UUID: 423b50ad-b5ee-4f4c-b6ee-c2474cfb72a0
Value: 28
Property: Read | Notify

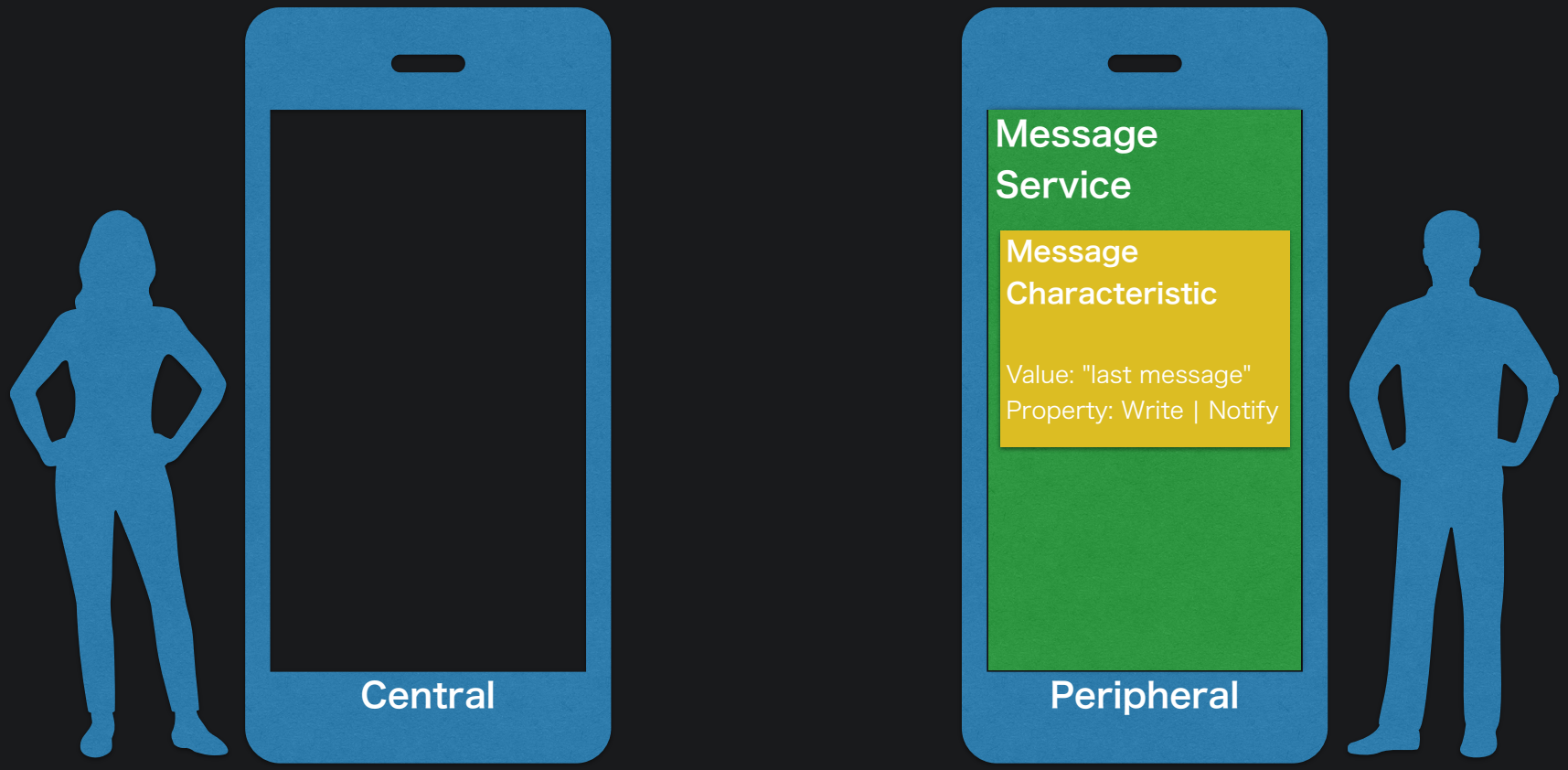
Characteristic (eco mode)

UUID: c3555675-f9a4-4ed8-b3ed-6654adb3d2a8
Value: 0
Property: Write

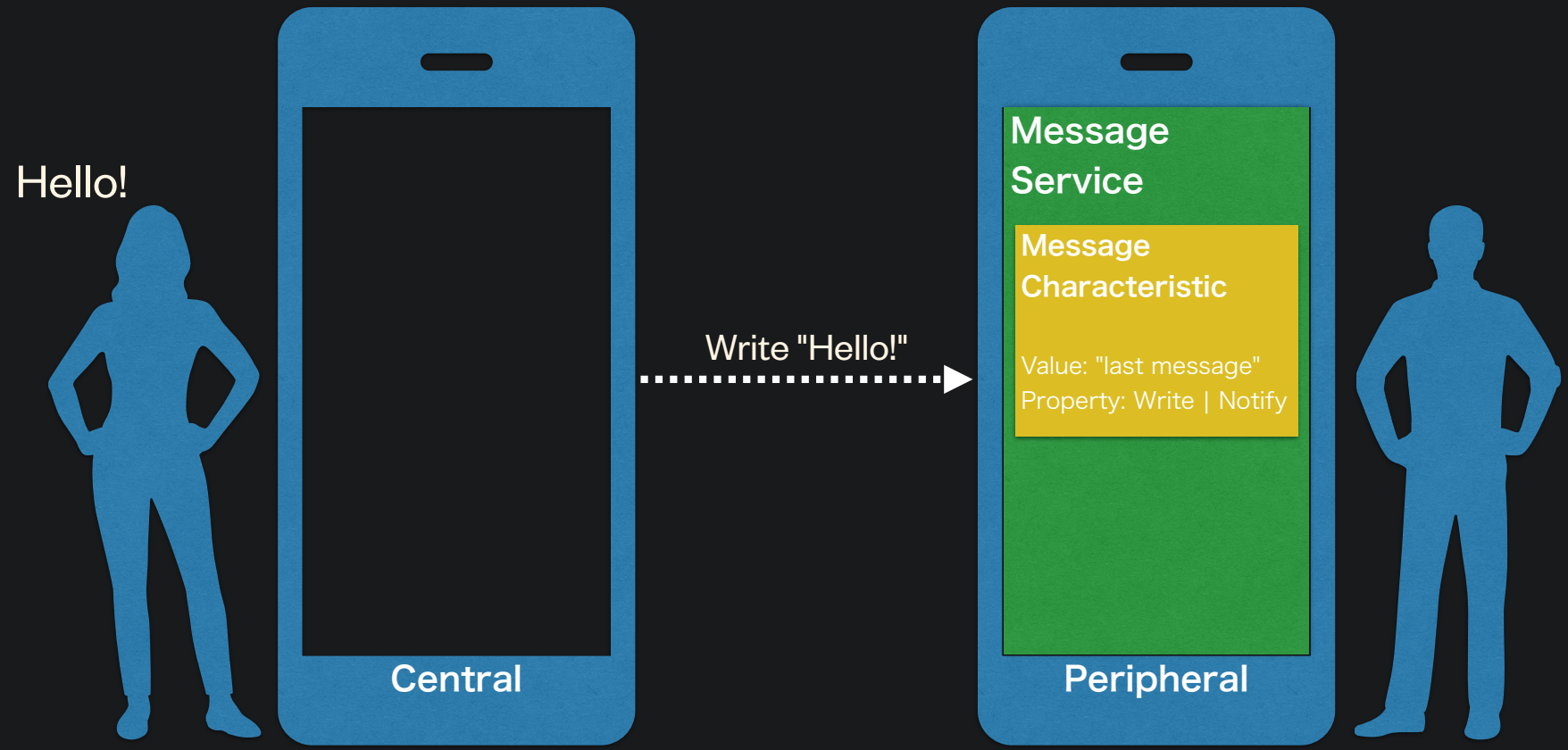
How it works on Android?

- Android can serve as Central with 4.3 or later
- Android can serve as Peripheral with 5.0 or later
- What's the point for Android to become Peripheral? I don't know🤷
  The idea is up to you!

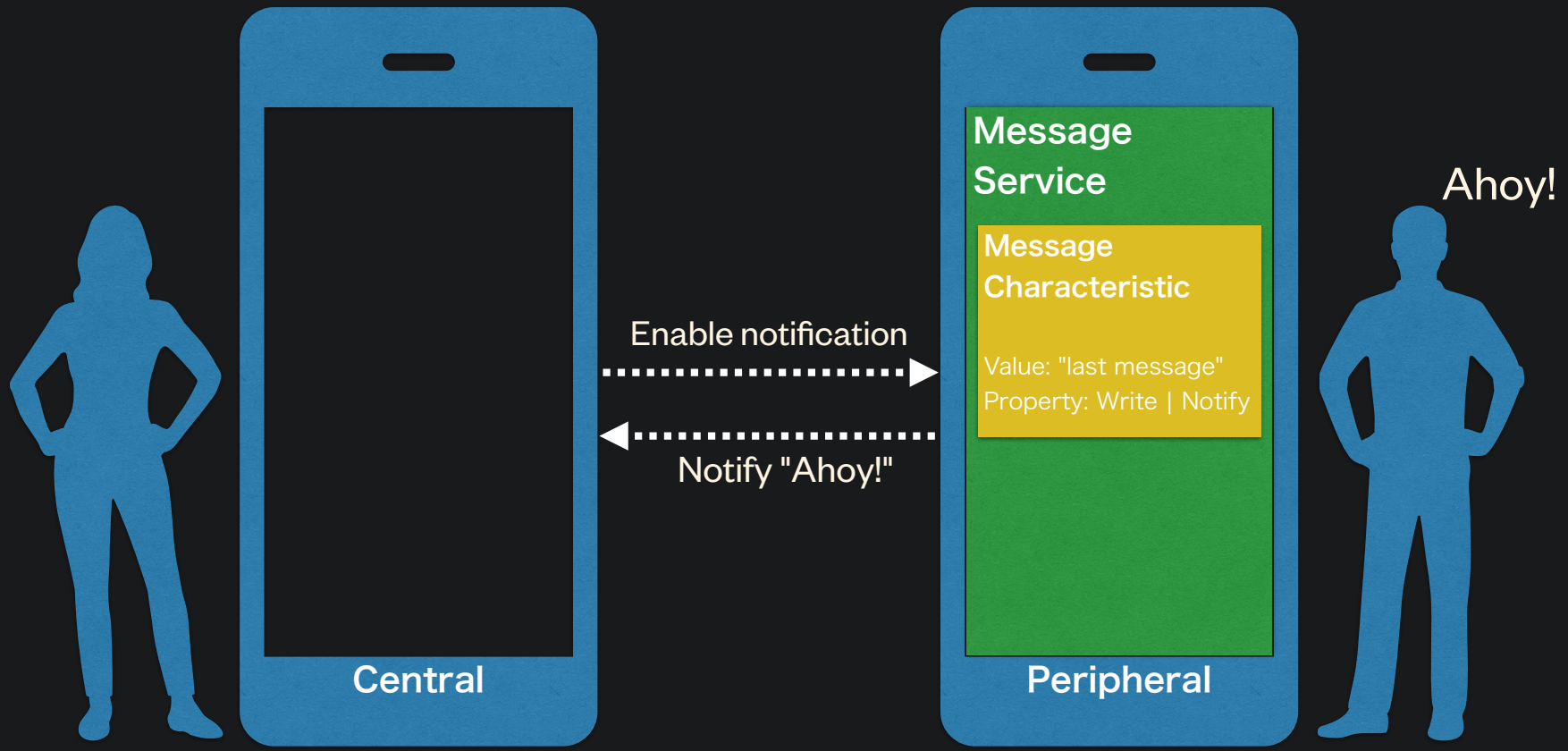
Two-way communication scenario



Two-way communication scenario



Two-way communication scenario



Permissions

```
<!-- Bluetooth Low Energy -->
<uses-permission
    android:name="android.permission.BLUETOOTH"
    android:maxSdkVersion="30" />
<uses-permission
    android:name="android.permission.BLUETOOTH_ADMIN"
    android:maxSdkVersion="30" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

<!-- Android 12 or higher -->
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
```

Check BLE support

```
if (!packageManager.hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
    // BLE is not supported on this device
    return
}

val bluetoothManager = getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
val bluetoothAdapter = bluetoothManager.adapter
if (bluetoothAdapter == null) {
    // Bluetooth is not supported on this device
    return
}
```

Central

1. Scan Peripherals around
2. Connect to Peripheral
3. Enable Notification

Central - Scan Peripheral

```
bluetoothLeScanner = bluetoothAdapter.bluetoothLeScanner
bluetoothLeScanner.startScan(scanCallback)
```

Central - Scan Peripheral

```
private val scanCallback: ScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)

        val bluetoothDevice = result.device
    }

    override fun onBatchScanResults(results: List<ScanResult>) {
        super.onBatchScanResults(results)
    }

    override fun onScanFailed(errorCode: Int) {
        super.onScanFailed(errorCode)
        Log.e(TAG, String.format("onScanFailed(errorCode = %d)", errorCode))
    }
}
```

Central - Scan Peripheral

```
private val scanCallback: ScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)

        val bluetoothDevice = result.device
    }

    override fun onBatchScanResults(results: List<ScanResult>) {
        super.onBatchScanResults(results)
    }

    override fun onScanFailed(errorCode: Int) {
        super.onScanFailed(errorCode)
        Log.e(TAG, String.format("onScanFailed(errorCode = %d)", errorCode))
    }
}
```

Central - Connect to Peripheral

```
fun connect(bluetoothDevice: BluetoothDevice) {
    bluetoothGatt = bluetoothDevice.connectGatt(activity, false, bluetoothGattCallback)
}
```

Central - Connect to Peripheral

```
private val bluetoothGattCallback: BluetoothGattCallback = object : BluetoothGattCallback() {
    override fun onPhyUpdate(...) { }
    override fun onPhyRead(...) { }
    override fun onConnectionStateChange(...) { }
    override fun onServicesDiscovered(...) { }
    override fun onCharacteristicRead(...) { }
    override fun onCharacteristicWrite(...) { }
    override fun onCharacteristicChanged(...) { }
    override fun onDescriptorRead(...) { }
    override fun onDescriptorWrite(...) { }
    override fun onReliableWriteCompleted(...) { }
    override fun onReadRemoteRssi(...) { }
    override fun onMtuChanged(...) { }
}
```

Central - Connect to Peripheral

```
override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
    super.onConnectionStateChange(gatt, status, newState)
    when (status) {
        BluetoothProfile.STATE_CONNECTED -> {
            val discoverServicesResult = gatt.discoverServices()
            ...
        }
        BluetoothProfile.STATE_DISCONNECTED -> {
            if (bluetoothGatt != null) {
                bluetoothGatt!!.close()
                bluetoothGatt = null
            }
            isConnected = false
        }
    }
}
```

Central - Connect to Peripheral

```
override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
    super.onConnectionStateChange(gatt, status, newState)
    when (status) {
        BluetoothProfile.STATE_CONNECTED -> {
            val discoverServicesResult = gatt.discoverServices()
            ...
        }
        BluetoothProfile.STATE_DISCONNECTED -> {
            if (bluetoothGatt != null) {
                bluetoothGatt!!.close()
                bluetoothGatt = null
            }
            isConnected = false
        }
    }
}
```

Central - Connect to Peripheral

```
override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
    super.onServicesDiscovered(gatt, status)
    if (status != BluetoothGatt.GATT_SUCCESS) {
        return
    }
    val service = gatt.getService(UUID_SERVICE)
    if (service == null) {
        return
    }
    val characteristic = service.getCharacteristic(UUID_CHARACTERISTIC)
    if (characteristic == null) {
        return
    }
    val descriptor = characteristic.getDescriptor(UUID_DESCRIPTOR)
    if (descriptor == null) {
        return
    }
```

Central - Connect to Peripheral

```
override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
    super.onServicesDiscovered(gatt, status)
    if (status != BluetoothGatt.GATT_SUCCESS) {
        return
    }
    val service = gatt.getService(UUID_SERVICE)
    if (service == null) {
        return
    }
    val characteristic = service.getCharacteristic(UUID_CHARACTERISTIC)
    if (characteristic == null) {
        return
    }
    val descriptor = characteristic.getDescriptor(UUID_DESCRIPTOR)
    if (descriptor == null) {
        return
    }
```

Central - Connect to Peripheral

```
override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
    super.onServicesDiscovered(gatt, status)
    if (status != BluetoothGatt.GATT_SUCCESS) {
        return
    }
    val service = gatt.getService(UUID_SERVICE)
    if (service == null) {
        return
    }
    val characteristic = service.getCharacteristic(UUID_CHARACTERISTIC)
    if (characteristic == null) {
        return
    }
    val descriptor = characteristic.getDescriptor(UUID_DESCRIPTOR)
    if (descriptor == null) {
        return
    }
```

Central - Connect to Peripheral

```
bluetoothGatt = gatt
bluetoothGattCharacteristic = characteristic
bluetoothLeScanner.stopScan(scanCallback)
```

Central - Enable Notification

```
val setCharacteristicNotificationResult = bluetoothGatt.setCharacteristicNotification(characteristic, true)
if (!setCharacteristicNotificationResult) {
    return
}

val descriptorSetValueResult = descriptor.setValue(BluetoothGattDescriptor.ENABLE_INDICATION_VALUE)
if (!descriptorSetValueResult) {
    return
}

val writeDescriptorResult = bluetoothGatt.writeDescriptor(descriptor)
if (!writeDescriptorResult) {
    return
}
```

Central - Enable Notification

```
val setCharacteristicNotificationResult = bluetoothGatt.setCharacteristicNotification(characteristic, true)
if (!setCharacteristicNotificationResult) {
    return
}

val descriptorSetValueResult = descriptor.setValue(BluetoothGattDescriptor.ENABLE_INDICATION_VALUE)
if (!descriptorSetValueResult) {
    return
}

val writeDescriptorResult = bluetoothGatt.writeDescriptor(descriptor)
if (!writeDescriptorResult) {
    return
}
```

Central - Enable Notification

```
val setCharacteristicNotificationResult = bluetoothGatt.setCharacteristicNotification(characteristic, true)
if (!setCharacteristicNotificationResult) {
    return
}

val descriptorSetValueResult = descriptor.setValue(BluetoothGattDescriptor.ENABLE_INDICATION_VALUE)
if (!descriptorSetValueResult) {
    return
}

val writeDescriptorResult = bluetoothGatt.writeDescriptor(descriptor)
if (!writeDescriptorResult) {
    return
}
```

Central - Enable Notification

```
override fun onCharacteristicChanged(gatt: BluetoothGatt,
                                     characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val message = characteristic.getStringValue(0)
        ...
    }
}
```

Central - Enable Notification

```
override fun onCharacteristicChanged(gatt: BluetoothGatt,
                                     characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val message = characteristic.getStringValue(0)
        ...
    }
}
```

Peripheral - advertisement

1. Open GATT Server
2. Start advertising

Peripheral - Open GATT Server

```
val bluetoothGattDescriptor = BluetoothGattDescriptor(
    UUID_DESCRIPTOR,
    BluetoothGattDescriptor.PERMISSION_READ or BluetoothGattDescriptor.PERMISSION_WRITE
)

val bluetoothGattCharacteristic = BluetoothGattCharacteristic(
    UUID_CHARACTERISTIC,
    BluetoothGattCharacteristic.PROPERTY_READ or BluetoothGattCharacteristic.PROPERTY_WRITE or
    BluetoothGattCharacteristic.PROPERTY_NOTIFY,
    BluetoothGattDescriptor.PERMISSION_WRITE or BluetoothGattCharacteristic.PERMISSION_READ
)
```

Peripheral - Open GATT Server

```
bluetoothGattCharacteristic.addDescriptor(bluetoothGattDescriptor)
val bluetoothGattService =
    BluetoothGattService(UUID_SERVICE, BluetoothGattService.SERVICE_TYPE_PRIMARY)
val addCharacteristicResult = bluetoothGattService.addCharacteristic(bluetoothGattCharacteristic)
if (!addCharacteristicResult) {
    return false
}

bluetoothGattServer = bluetoothManager.openGattServer(activity, bluetoothGattServerCallback)
val addServiceResult = bluetoothGattServer.addService(bluetoothGattService)
if (!addServiceResult) {
    return false
}
return true
```

Peripheral - Open GATT Server

```
bluetoothGattCharacteristic.addDescriptor(bluetoothGattDescriptor)
val bluetoothGattService =
    BluetoothGattService(UUID_SERVICE, BluetoothGattService.SERVICE_TYPE_PRIMARY)
val addCharacteristicResult = bluetoothGattService.addCharacteristic(bluetoothGattCharacteristic)
if (!addCharacteristicResult) {
    return false
}

bluetoothGattServer = bluetoothManager.openGattServer(activity, bluetoothGattServerCallback)
val addServiceResult = bluetoothGattServer.addService(bluetoothGattService)
if (!addServiceResult) {
    return false
}
return true
```

Peripheral - Open GATT Server

```
private val bluetoothGattServerCallback: BluetoothGattServerCallback
        = object : BluetoothGattServerCallback() {
    override fun onConnectionStateChange(...) { }
    override fun onServiceAdded(...) { }
    override fun onCharacteristicReadRequest(...) { }
    override fun onCharacteristicWriteRequest(...) { }
    override fun onDescriptorReadRequest(...) { }
    override fun onDescriptorWriteRequest(...) { }
    override fun onExecuteWrite(...) { }
    override fun onNotificationSent(...) { }
    override fun onMtuChanged(...) { }
    override fun onPhyUpdate(...) { }
    override fun onPhyRead(...) { }
}
```

Peripheral - Open GATT Server

```
override fun onConnectionStateChange(device: BluetoothDevice, status: Int, newState: Int) {
    super.onConnectionStateChange(device, status, newState)
    when (status) {
        BluetoothProfile.STATE_CONNECTED -> {
            bluetoothLeAdvertiser.stopAdvertising(advertiseCallback)
            remoteDevice = device
            isConnected = true
        }
        BluetoothProfile.STATE_DISCONNECTED -> {
            isConnected = false
            remoteDevice = null
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onConnectionStateChange(device: BluetoothDevice, status: Int, newState: Int) {
    super.onConnectionStateChange(device, status, newState)
    when (status) {
        BluetoothProfile.STATE_CONNECTED -> {
            bluetoothLeAdvertiser.stopAdvertising(advertiseCallback)
            remoteDevice = device
            isConnected = true
        }
        BluetoothProfile.STATE_DISCONNECTED -> {
            isConnected = false
            remoteDevice = null
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onDescriptorWriteRequest(
    device: BluetoothDevice, requestId: Int, descriptor: BluetoothGattDescriptor,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    super.onDescriptorWriteRequest(
        device, requestId, descriptor, preparedWrite, responseNeeded, offset, value )
    if (UUID_DESCRIPTOR == descriptor.uuid) {
        if (responseNeeded) {
            bluetoothGattServer.sendResponse(
                device, requestId, BluetoothGatt.GATT_SUCCESS, offset, value
            )
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onDescriptorWriteRequest(
    device: BluetoothDevice, requestId: Int, descriptor: BluetoothGattDescriptor,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    super.onDescriptorWriteRequest(
        device, requestId, descriptor, preparedWrite, responseNeeded, offset, value )
    if (UUID_DESCRIPTOR == descriptor.uuid) {
        if (responseNeeded) {
            bluetoothGattServer.sendResponse(
                device, requestId, BluetoothGatt.GATT_SUCCESS, offset, value
            )
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onDescriptorWriteRequest(
    device: BluetoothDevice, requestId: Int, descriptor: BluetoothGattDescriptor,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    super.onDescriptorWriteRequest(
        device, requestId, descriptor, preparedWrite, responseNeeded, offset, value )
    if (UUID_DESCRIPTOR == descriptor.uuid) {
        if (responseNeeded) {
            bluetoothGattServer.sendResponse(
                device, requestId, BluetoothGatt.GATT_SUCCESS, offset, value
            )
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onCharacteristicWriteRequest(
    device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    super.onCharacteristicWriteRequest(
        device, requestId, characteristic, preparedWrite,
        responseNeeded, offset, value)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        try {
            ...
        } finally {
            if (responseNeeded) {
                bluetoothGattServer.sendResponse(
                    device, requestId, BluetoothGatt.GATT_SUCCESS, offset, value)
            }
        }
    }
}
```

Peripheral - Open GATT Server

```
override fun onCharacteristicWriteRequest(
    device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    super.onCharacteristicWriteRequest(
        device, requestId, characteristic, preparedWrite,
        responseNeeded, offset, value)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        try {
            ...
        } finally {
            if (responseNeeded) {
                bluetoothGattServer.sendResponse(
                    device, requestId, BluetoothGatt.GATT_SUCCESS, offset, value)
            }
        }
    }
}
```

Peripheral - Open GATT Server

```
if (UUID_CHARACTERISTIC == characteristic.uuid) {
    try {
        val setValueResult = characteristic.setValue(value)
        if (!setValueResult) {
            return
        }
        val message = characteristic.getStringValue(0)
    } finally {
        ...
    }
}
```

Peripheral - Open GATT Server

```
if (UUID_CHARACTERISTIC == characteristic.uuid) {
    try {
        val setValueResult = characteristic.setValue(value)
        if (!setValueResult) {
            return
        }
        val message = characteristic.getStringValue(0)
    } finally {
        ...
    }
}
```

Peripheral - Start advertising

```
val settingsBuilder = AdvertiseSettings.Builder().apply {
    setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_BALANCED)
    setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_MEDIUM)
    setTimeout(TIMEOUT)
    setConnectable(true)
}
val dataBuilder = AdvertiseData.Builder().apply {
    setIncludeTxPowerLevel(true)
    addServiceUuid(ParcelUuid.fromString(UUID_SERVICE.toString()))
}
val responseBuilder = AdvertiseData.Builder().apply {
    setIncludeDeviceName(true)
}
```

Peripheral - Start advertising

```
val bluetoothLeAdvertiser: BluetoothLeAdvertiser = bluetoothAdapter.bluetoothLeAdvertiser
    ?: // This device does not support BLE Peripheral mode.
    return
bluetoothLeAdvertiser.startAdvertising(
    settingsBuilder.build(),
    dataBuilder.build(),
    responseBuilder.build(),
    advertiseCallback
)
```

Peripheral - Start advertising

```
private val advertiseCallback: AdvertiseCallback = object : AdvertiseCallback() {
    override fun onStartSuccess(settingsInEffect: AdvertiseSettings) {
        Log.d(TAG, "BLE advertising success: $settingsInEffect")
    }

    override fun onStartFailure(errorCode: Int) {
        Log.e(TAG, "BLE advertising failure. errorCode: $errorCode")
    }
}
```

Two-way communication

Two-way communication

```
fun write(message: String) {
    if (isPeripheral) {
        // Send message via Notification
    } else {
        // Send message by Write operation
    }
}
```

Peripheral - write

```
val setValueResult = bluetoothGattCharacteristic.setValue(message)
if (!setValueResult) {
    return false
}

val notificationResult = bluetoothGattServer.notifyCharacteristicChanged(
        remoteDevice,
        bluetoothGattCharacteristic,
        true)

if (!notificationResult) {
    return false
}
```

Peripheral - write

```
val setValueResult = bluetoothGattCharacteristic.setValue(message)
if (!setValueResult) {
    return false
}

val notificationResult = bluetoothGattServer.notifyCharacteristicChanged(
        remoteDevice,
        bluetoothGattCharacteristic,
        true)

if (!notificationResult) {
    return false
}
```

Central - write

```
val setValueResult = bluetoothGattCharacteristic.setValue(telegramString)
if (!setValueResult) {
    return
}

val writeCharacteristicResult =
    bluetoothGatt.writeCharacteristic(bluetoothGattCharacteristic)
if (!writeCharacteristicResult) {
    return
}
```

Central - write

```
val setValueResult = bluetoothGattCharacteristic.setValue(telegramString)
if (!setValueResult) {
    return
}

val writeCharacteristicResult =
    bluetoothGatt.writeCharacteristic(bluetoothGattCharacteristic)
if (!writeCharacteristicResult) {
    return
}
```

Tips

- The content explained so far is the most basic scenario, and I intentionally dropped important concept, **MTU**
- Requests for larger MTU can be made easily, But there is **no guarantee that the request will be accepted**
- How to handle messages larger than MTU
  - Central approach
  - Peripheral approach

Request for larger MTU

Request for larger MTU

```
private const val MAX_MTU = 512

val requestMtuResult = bluetoothGatt.requestMtu(MAX_MTU)

override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {
    super.onMtuChanged(gatt, mtu, status)

    currentMTU = mtu
}
```

Handle messages larger than MTU - Central

Handle messages larger than MTU - Central

- Write from Central to Peripheral is relatively straight forward `bluetoothGatt.writeCharacteristic(bluetoothGattCharacteristic)`
- If you try to send a message with more than MTU, the Android framework will **automatically try to split the message**
- In other words, as long as you know how to do it right, this is not difficult to accomplish
- This is handled by `BluetoothGattServerCallback` on Peripheral side

Handle messages larger than MTU - Central

```
private val byteArrayOutputStream = ByteArrayOutputStream()

override fun onCharacteristicWriteRequest(
    device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    ...
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        try {
            if (!preparedWrite) {
                // keep the same logic as before
            } else {
                try {
                    byteArrayOutputStream.write(value)
                } catch (e: IOException) {
                    Log.e(TAG, e.message, e)
                }
            }
```

Handle messages larger than MTU - Central

```
private val byteArrayOutputStream = ByteArrayOutputStream()

override fun onCharacteristicWriteRequest(
    device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    ...
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        try {
            if (!preparedWrite) {
                // keep the same logic as before
            } else {
                try {
                    byteArrayOutputStream.write(value)
                } catch (e: IOException) {
                    Log.e(TAG, e.message, e)
                }
            }
```

Handle messages larger than MTU - Central

```
private val byteArrayOutputStream = ByteArrayOutputStream()

override fun onCharacteristicWriteRequest(
    device: BluetoothDevice, requestId: Int, characteristic: BluetoothGattCharacteristic,
    preparedWrite: Boolean, responseNeeded: Boolean, offset: Int, value: ByteArray) {
    ...
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        try {
            if (!preparedWrite) {
                // keep the same logic as before
            } else {
                try {
                    byteArrayOutputStream.write(value)
                } catch (e: IOException) {
                    Log.e(TAG, e.message, e)
                }
            }
```

Handle messages larger than MTU - Central

```
// The Callback called after all preparedWrites are finished

override fun onExecuteWrite(device: BluetoothDevice, requestId: Int, execute: Boolean) {
    super.onExecuteWrite(device, requestId, execute)

    val bytes = byteArrayOutputStream.toByteArray()
    byteArrayOutputStream.reset()
    val message = String(bytes, StandardCharsets.UTF_8)
    ...
}
```

Handle messages larger than MTU - Peripheral

Handle messages larger than MTU - Peripheral

- We are using Notification to send messages from Peripheral, but **Notification cannot send messages larger than MTU**
- Notification does not automatically split messages
- **We need to split and send messages on our own**
- The minimum MTU seems to be about 20-23 bytes, but write code so that it works even if it changes dynamically depending on the situation

Handle messages larger than MTU - Peripheral

- Let's define a protocol
  - The first byte is the total number of chunks
  - The next byte is the current index of the total chunk
  - The remaining bytes are the payload

Handle messages larger than MTU - Peripheral

```
private const val MIN_MTU = 20

// This will be overridden later when the exact value is known
private var currentMTU = MIN_MTU

override fun onMtuChanged(device: BluetoothDevice, mtu: Int) {
    super.onMtuChanged(device, mtu)
    currentMTU = mtu
}
```

Handle messages larger than MTU - Peripheral

```
private fun sendNotification(message: String): Boolean {
    val bytes = message.toByteArray(StandardCharsets.UTF_8)
    val byteSize = bytes.size
    val headerSize = 2
    val payloadSize = currentMTU - headerSize
    val chunks = (byteSize + payloadSize - 1) / payloadSize
    if (chunks > 1 shl 8) {

        return false
    }
    ...
```

Handle messages larger than MTU - Peripheral

```
for (i in 0 until chunks) {
    val srcPos = i * payloadSize
    val length = if (srcPos + payloadSize > byteSize) byteSize - srcPos else payloadSize
    val totalChunkByte = chunks.toByte()
    val currentChunkByte = (i + 1).toByte()
    val partialBytes = ByteArray(headerSize + length)
    partialBytes[0] = totalChunkByte
    partialBytes[1] = currentChunkByte
    System.arraycopy(bytes, srcPos, partialBytes, headerSize, length)
    val setValueResult = bluetoothGattCharacteristic.setValue(partialBytes)
    if (!setValueResult) {
        return false
    }
    val notificationResult = bluetoothGattServer.notifyCharacteristicChanged(
            remoteDevice,
            bluetoothGattCharacteristic,
            true)
    return notificationResult
```

Handle messages larger than MTU - Peripheral

```
for (i in 0 until chunks) {
    val srcPos = i * payloadSize
    val length = if (srcPos + payloadSize > byteSize) byteSize - srcPos else payloadSize
    val totalChunkByte = chunks.toByte()
    val currentChunkByte = (i + 1).toByte()
    val partialBytes = ByteArray(headerSize + length)
    partialBytes[0] = totalChunkByte
    partialBytes[1] = currentChunkByte
    System.arraycopy(bytes, srcPos, partialBytes, headerSize, length)
    val setValueResult = bluetoothGattCharacteristic.setValue(partialBytes)
    if (!setValueResult) {
        return false
    }
    val notificationResult = bluetoothGattServer.notifyCharacteristicChanged(
            remoteDevice,
            bluetoothGattCharacteristic,
            true)
    return notificationResult
```

Handle messages larger than MTU - Peripheral

```
// Central
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        ...
    }
}
```

Handle messages larger than MTU - Peripheral

```
private val byteArrayOutputStream = ByteArrayOutputStream()
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val bytes = characteristic.value
        val totalChunkByte = bytes[0]
        val currentChunkByte = bytes[1]
        val headerSize = 2
        val byteSize = bytes.size
        val buffer = ByteArray(byteSize - headerSize)
        System.arraycopy(bytes, headerSize, buffer, 0, byteSize - headerSize)
        val totalChunk = java.lang.Byte.toUnsignedInt(totalChunkByte)
        val currentChunk = java.lang.Byte.toUnsignedInt(currentChunkByte)
        try {
            byteArrayOutputStream.write(buffer)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
        if (totalChunk == currentChunk) {
            val fullMessage = byteArrayOutputStream.toString()
            fullMessage.length
            byteArrayOutputStream.reset()
        }
    }
}
```

Handle messages larger than MTU - Peripheral

```
private val byteArrayOutputStream = ByteArrayOutputStream()
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val bytes = characteristic.value
        val totalChunkByte = bytes[0]
        val currentChunkByte = bytes[1]
        val headerSize = 2
        val byteSize = bytes.size
        val buffer = ByteArray(byteSize - headerSize)
        System.arraycopy(bytes, headerSize, buffer, 0, byteSize - headerSize)
        val totalChunk = java.lang.Byte.toUnsignedInt(totalChunkByte)
        val currentChunk = java.lang.Byte.toUnsignedInt(currentChunkByte)
        try {
            byteArrayOutputStream.write(buffer)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
        if (totalChunk == currentChunk) {
            val fullMessage = byteArrayOutputStream.toString()
            fullMessage.length
            byteArrayOutputStream.reset()
        }
    }
}
```

Handle messages larger than MTU - Peripheral

```
private val byteArrayOutputStream = ByteArrayOutputStream()
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val bytes = characteristic.value
        val totalChunkByte = bytes[0]
        val currentChunkByte = bytes[1]
        val headerSize = 2
        val byteSize = bytes.size
        val buffer = ByteArray(byteSize - headerSize)
        System.arraycopy(bytes, headerSize, buffer, 0, byteSize - headerSize)
        val totalChunk = java.lang.Byte.toUnsignedInt(totalChunkByte)
        val currentChunk = java.lang.Byte.toUnsignedInt(currentChunkByte)
        try {
            byteArrayOutputStream.write(buffer)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
        if (totalChunk == currentChunk) {
            val fullMessage = byteArrayOutputStream.toString()
            fullMessage.length
            byteArrayOutputStream.reset()
        }
    }
}
```

Handle messages larger than MTU - Peripheral

```
private val byteArrayOutputStream = ByteArrayOutputStream()
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic: BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    if (UUID_CHARACTERISTIC == characteristic.uuid) {
        val bytes = characteristic.value
        val totalChunkByte = bytes[0]
        val currentChunkByte = bytes[1]
        val headerSize = 2
        val byteSize = bytes.size
        val buffer = ByteArray(byteSize - headerSize)
        System.arraycopy(bytes, headerSize, buffer, 0, byteSize - headerSize)
        val totalChunk = java.lang.Byte.toUnsignedInt(totalChunkByte)
        val currentChunk = java.lang.Byte.toUnsignedInt(currentChunkByte)
        try {
            byteArrayOutputStream.write(buffer)
        } catch (e: IOException) {
            Log.e(TAG, e.message, e)
        }
        if (totalChunk == currentChunk) {
            val fullMessage = byteArrayOutputStream.toString()
            fullMessage.length
            byteArrayOutputStream.reset()
        }
    }
}
```

Tips

- The most important thing with BLE is to ensure that all operations are sequential
- Read the official site documentation properly and make sure you return the response you need
- Avoiding the infamous **GATT_ERROR(133)** is not a matter of needless retries or unfounded SLEEP!

How to use each one differently

How to use each one differently

- When should Wi-Fi Direct be used?
- When should Bluetooth Classic be used?
- When should BLE be used?

When to use Wi-Fi Direct

Q. When should Wi-Fi Direct be used?

A. When sending and receiving large files

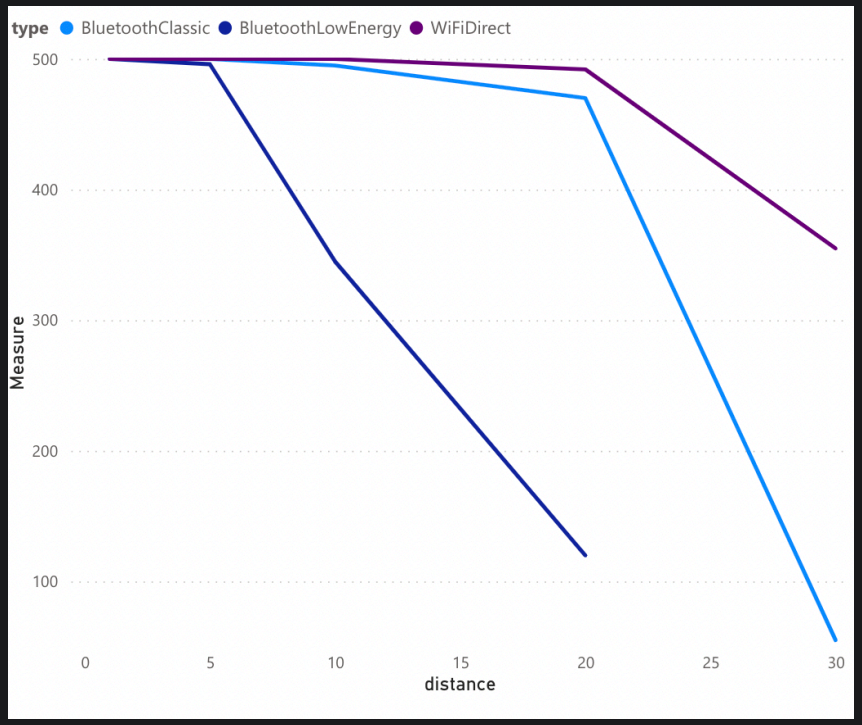
When to use Wi-Fi Direct

- **TL;DR** Wi-Fi is **incomparably** faster!
  - Between two Pixel 3a XL
  - Distance between the two units is 50cm
  - Average of 50 A to B and 50 B to A runs each (total 100 runs)

	Wi-Fi Direct	Bluetooth Classic
Transfer 1MB file	15.86MB/ Sec	115.57KB/ Sec

When to use Wi-Fi Direct

- Another set of data shows that Wi-Fi has a high packet reachability rate over long distances
  - Between Pixel 3a XL and BLU90
  - Distance between the two units is 1, 5, 10, 20, and 30m
  - The graph shows the success rate of packet transmission at each distance



When to use Bluetooth Classic

Q. When should Bluetooth Classic be used?

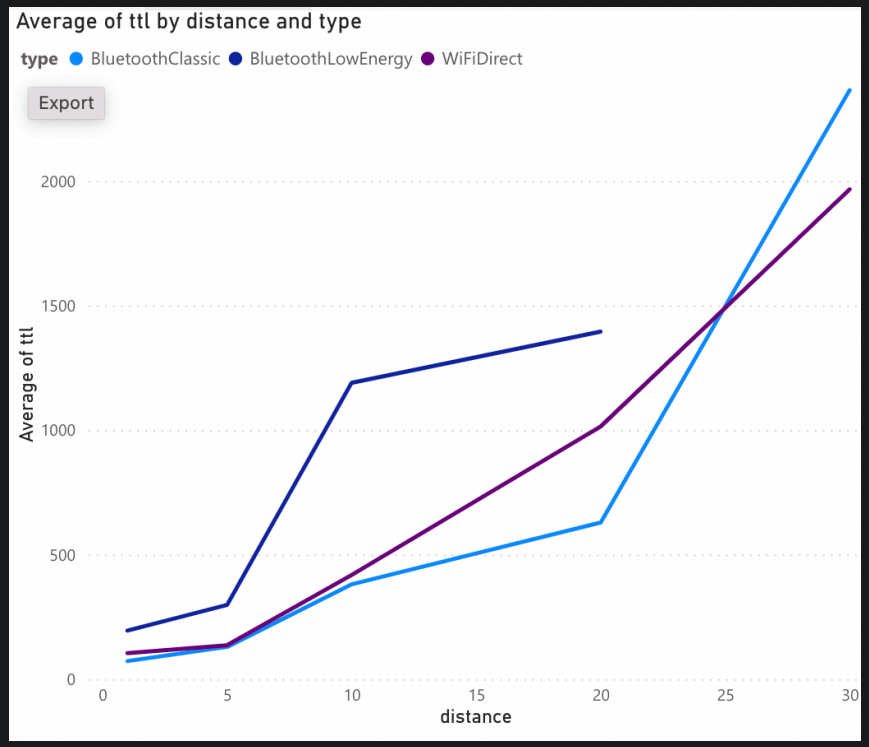
A. Situations other than sending large files

When to use Bluetooth Classic

- Studies show that Bluetooth frequency hopping is resistant to network congestion
- Decent pairing speed. Wi-Fi Direct takes a few seconds to a dozen seconds before they can connect to each other, while Bluetooth Classic takes 1-2 seconds or less

When to use Bluetooth Classic

- In my experiments, Bluetooth Classic had the fastest packet arrival time over short to medium distances
  - Between Pixel 3a XL and BLU90
  - Distance between the two units is 1, 5, 10, 20, and 30m
  - The graph shows the TTL of packet arrival at each distance. The lower the line is located, the faster it is



When to use Bluetooth Low Energy

Q. When should BLE be used?

A. 🤷 I'd love to hear your idea

Steps to more advanced topics

Nearby Connections API

- https://developers.google.com/nearby/connections/overview
- *“Nearby Connections is a peer-to-peer networking API that allows apps to easily discover, connect to, and exchange data with nearby devices in real-time, regardless of network connectivity”*
- *It supports multiple network topologies such as peer-to-peer, star, and mesh, depending on the purpose*
- *Looks like the kind of wireless communication we talked about in today's talk will be used under the hood.*

Any Questions?

Thanks!