Lab on apps development for tablets, smartphones and smartwatches

Week 8: Connectivity - Interacting with peripherals

Dr. Marina Zapater, Prof. David Atienza

Mr. Grégoire Surrel, Ms. Elisabetta de Giovanni, Mr. Dionisijie Sopic, Ms. Halima Najibi

Embedded Systems Laboratory (ESL) – Faculty of Engineering (STI)

Outline of the class

- **Connecting to the internet: WiFi and 3G/4G**
  - Transferring data without draining your battery

- Bluetooth
  - Bluetooth Low Energy
  - Polar H7

- NFC → slides only

Connecting to the Internet

1. Add permissions to Android Manifest
2. Check Network Connection
3. Create Worker Thread
4. Implement background task
   - Create URI
   - Make HTTP Connection
   - Connect and GET Data
5. Process results
   - Parse Results

1. Manifest.xml:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

2. Using ConnectivityManager and NetworkInfo

- ConnectivityManager:
  - Answers queries about the state of network
  - Notifies app when network connectivity changes
- NetworkInfo:
  - Describes status of a network interface
  - Mobile or WiFi

Connecting to the Internet (1)

1. Add permissions to Android Manifest
2. Check Network Connection
3. Create Worker Thread
4. Implement background task
   - Create URI
   - Make HTTP Connection
   - Connect and GET Data
5. Process results
   - Parse Results

1. Manifest.xml:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

2. Using ConnectivityManager and NetworkInfo

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        TextView textView = (TextView) findViewById(R.id.textView);

        ConnectivityManager connMgr = (ConnectivityManager)
                getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

        if (networkInfo != null && networkInfo.isConnected()) {
            //Check type of network connectivity
            if (networkInfo.getType() == ConnectivityManager.TYPE_WIFI){
                textView.setText("Connected to Wifi");
            }
            else if (networkInfo.getType() == ConnectivityManager.TYPE_MOBILE){
                textView.setText("Connected to Mobile network");
            }
            // Create background thread to connect and get data
            String stringUrl = "hello";
            new DownloadWebpageTask().execute(stringUrl);
        } else {
            textView.setText("No network connection available.");
        }

    }
}
```

3.

Connecting to the Internet (2)

4. Implement background task
   - doInBackground()
     - Create URI
     - Make HTTP Connection
     - Connect and GET Data
5. Process results
   - Parse Results! → onPostExecute()
   - See next slide!

4. We can use: AsyncTask, AsyncTaskLoader, Service

```
class DownloadWebpageTask extends AsyncTask<String,Void,String>{

    final String BASE_URL =
            "https://www.googleapis.com/books/v1/volumes?";
    final String QUERY_PARAM = "q";
    final String MAX_RESULTS = "maxResults";
    final String PRINT_TYPE = "printType";

    @Override
    protected String doInBackground(String... strings) {

        try {
            //Create URI
            Uri builtURI = Uri.parse(BASE_URL).buildUpon()
                .appendQueryParameter(QUERY_PARAM, String.valueOf(strings))
                .appendQueryParameter(MAX_RESULTS, "10")
                .appendQueryParameter(PRINT_TYPE, "books")
                .build();
            URL requestURL= new URL(builtURI.toString());

            //Make HTTP Connection
            HttpURLConnection conn = (HttpURLConnection) requestURL.openConnection();
            conn.setReadTimeout(10000 /* milliseconds */);
            conn.setConnectTimeout(15000 /* milliseconds */);
            conn.setRequestMethod("GET");
            conn.setDoInput(true);

            //Connect and GET Data
            conn.connect();
            int response = conn.getResponseCode();

            InputStream is = conn.getInputStream();
            String contentAsString = is.toString();
            return contentAsString;

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

Parsing the results (Implementing Step 5)

- Usually, response is in JSON or XML
  - Use helper classes!
    - JSONObject, JSONArray
    - XMLPullParser
- What is JSON?
  - **J**ava**S**cript **O**bject **N**otation
  - Syntax for storing and exchanging data
    - Data is in name/value pairs
    - Data is separated by commas
    - Curly braces hold objects
    - Square brackets hold arrays

XML Example

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON Example

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON object example

- Implement method to receive and handle results
  - AsyncTask → onPostExecute()
- Using JSONObject and JSONArray Java methods
  - Parsing the JSON file

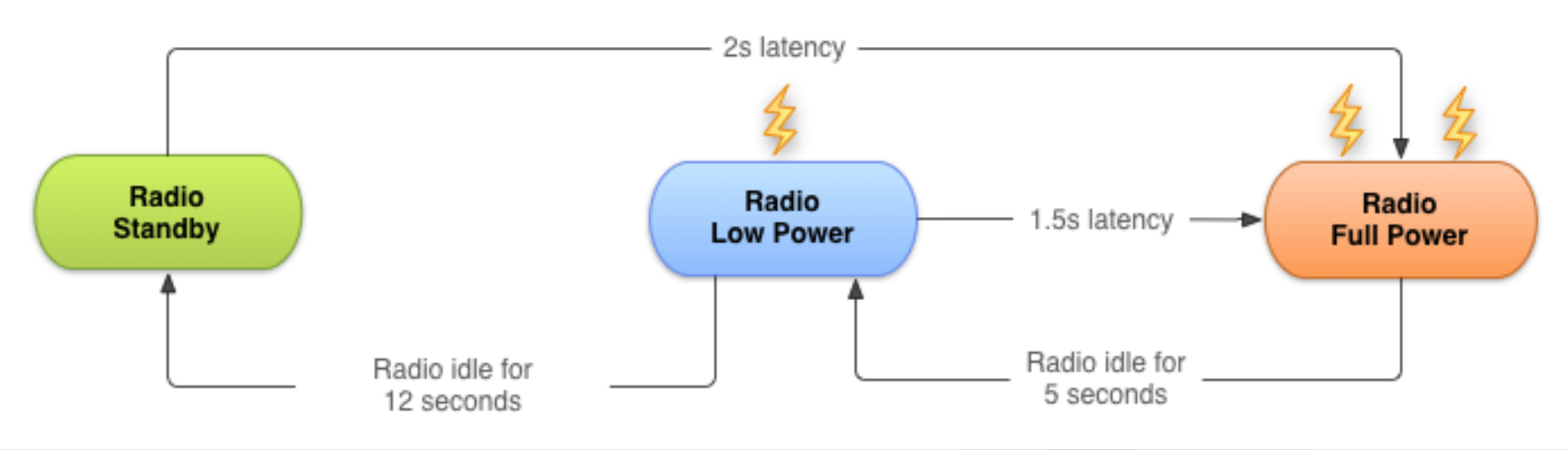
Example: simple JSON file

```
{
   "population":1,252,000,000,
   "country":"India",
   "cities":["NewDelhi","Mumbai","Kolkata"]
}
```

```java
@Override
protected void onPostExecute(String response) {
    super.onPostExecute(response);

    try {
        JSONObject jsonObject = new JSONObject(response);
        String nameOfCountry = (String) jsonObject.get("country");
        long population = (Long) jsonObject.get("population");
        JSONArray listOfCities = (JSONArray) jsonObject.get("cities");
        for (int i=0; i<listOfCities.length(); i++){
            // do something
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Transferring data efficiently

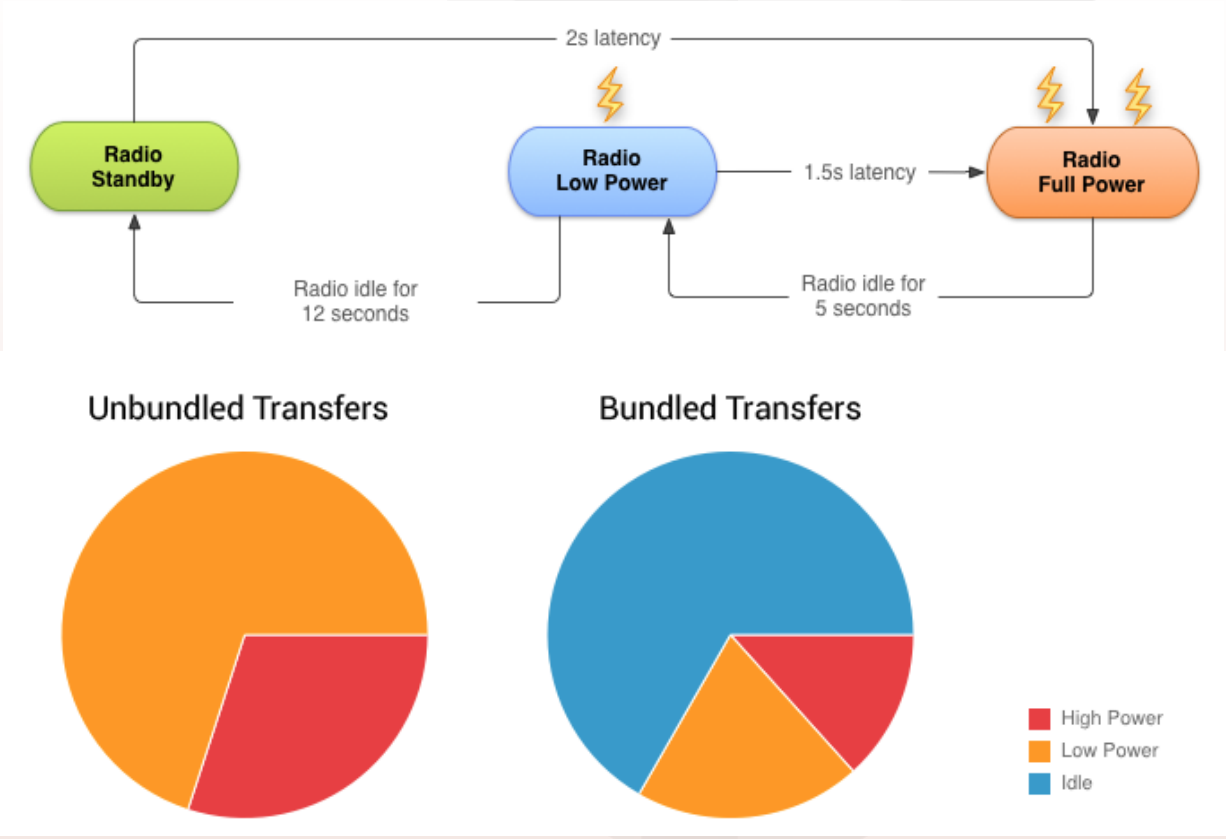
- Transferring data uses resources:
  - Wireless radio uses battery:
    - Device runs out of battery → Need to let device charge
  - Wireless radio power states:
    - Full power—Active connection, highest rate data transfer
    - Low power—Intermediate state that uses 50% less power
    - Standby—Minimal energy, no active network connection



Solution: bundle network transfers!

- For a typical 3G device, every data transfer session, the radio draws energy for almost 20 seconds
  - Send data for 1s every 18s—radio mostly on full power
  - Send data in bundles of 3s—radio mostly idle

- Prefetch data:
  - Download all the data you are likely to need in a single burst, over a single connection, at full capacity
  - If you guess right, reduces battery cost and latency



Monitor connectivity and battery state

- Transferring data uses up data plans:
  - Costing users real money (for free apps…)

- Monitoring connectivity:
  - WiFi radio uses less battery and has more bandwidth than wireless radio
  - Use ConnectivityManager to determine which radio is active and adapt your strategy

- Monitoring battery:
  - Wait for specific conditions to initiate battery intensive operation
  - BatteryManager broadcasts all battery and charging details

Outline of the class

- Connecting to the internet: WiFi and 3G/4G
  - Transferring data without draining your battery

- **Bluetooth**
  - Bluetooth Low Energy
  - Polar H7

- NFC → slides only

Bluetooth

- Android includes Bluetooth network stack and APIs to:
  - Scan for other Bluetooth devices
  - Query the local Bluetooth adapter for paired Bluetooth devices
  - Connect to other devices through service discovery
  - Transfer data to and from other devices
  - Manage multiple connections

- Profiles: starting on Android 3.0, specific support for certain devices
  - Headset → BluetoothHeadset class
  - A2DP (Advanced Audio Distribution Profile) → BluetoothA2dp class
  - Health device → Android 4.0, Bluetooth Health Device Profile (HDP)
    - heart-rate monitors, blood meters, thermometers, scales…

Bluetooth Implementation steps (1)

1. Setting up Bluetooth
   a) Permissions
   b) Getting the Bluetooth adapter
   c) Enabling bluetooth
      - A dialog will appear requesting user permission to use Bluetooth
2. Finding devices
3. Connecting devices
4. Managing a Connection

1.a)

```
<manifest ... >
  <uses-permission android:name="android.permission.BLUETOOTH" />
  <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  ...
</manifest>
```

1.b)

```
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
    // Device does not support Bluetooth
}
```

1.c)

```
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

Bluetooth Implementation steps (2)

1. Setting up Bluetooth
2. Finding devices
   a) Querying the list of paired devices
   b) Or… via Device Discovery
      - Using StartDiscovery
3. Connecting devices
4. Managing a Connection

2.a) Querying the paired devices

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();

if (pairedDevices.size() > 0) {
    // There are paired devices. Get the name and address of each paired device.
    for (BluetoothDevice device : pairedDevices) {
        String deviceName = device.getName();
        String deviceHardwareAddress = device.getAddress(); // MAC address
    }
}
```

2.b) Discovering new devices

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...

    // Register for broadcasts when a device is discovered.
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(mReceiver, filter);
}

// Create a BroadcastReceiver for ACTION_FOUND.
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            // Discovery has found a device. Get the BluetoothDevice
            // object and its info from the Intent.
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            String deviceName = device.getName();
            String deviceHardwareAddress = device.getAddress(); // MAC address
        }
    }
};

@Override
protected void onDestroy() {
    super.onDestroy();
    ...

    // Don't forget to unregister the ACTION_FOUND receiver.
    unregisterReceiver(mReceiver);
}
```

Bluetooth Implementation steps (3)

1. Setting up Bluetooth
2. Finding devices
3. Connecting devices
   a) Connecting as a server:
      - Getting a BluetoothServerSocket
      - Listening for incoming connections using accept()
   b) Connecting as a client:
      - Get a BluetoothSocket
      - Connect to it using connect()
   - You need a UUID:
      - 128-bit number that uniquely identifies the Bluetooth device
4. Managing a Connection

3.a) Connecting as a server

```
private class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {
        // Use a temporary object that is later assigned to mmServerSocket
        // because mmServerSocket is final.
        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code.
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) {
            Log.e(TAG, "Socket's listen() method failed", e);
        }
        mmServerSocket = tmp;
    }

    public void run() {
        BluetoothSocket socket = null;
        // Keep listening until exception occurs or a socket is returned.
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                Log.e(TAG, "Socket's accept() method failed", e);
                break;
            }

            if (socket != null) {
                // A connection was accepted. Perform work associated with
                // the connection in a separate thread.
                manageMyConnectedSocket(socket);
                mmServerSocket.close();
                break;
            }
        }
    }
```

Bluetooth Implementation steps (3)

1. Setting up Bluetooth
2. Finding devices
3. Connecting devices
   a) Connecting as a server:
      - Getting a BluetoothServerSocket
      - Listening for incoming connections using accept()
   b) Connecting as a client:
      - Get a BluetoothSocket
      - Connect to it using connect()
4. Managing a Connection

3.b) Connecting as a client

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        // Use a temporary object that is later assigned to mmSocket
        // because mmSocket is final.
        BluetoothSocket tmp = null;
        mmDevice = device;

        try {
            // Get a BluetoothSocket to connect with the given BluetoothDevice.
            // MY_UUID is the app's UUID string, also used in the server code.
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) {
            Log.e(TAG, "Socket's create() method failed", e);
        }
        mmSocket = tmp;
    }

    public void run() {
        // Cancel discovery because it otherwise slows down the connection.
        mBluetoothAdapter.cancelDiscovery();

        try {
            // Connect to the remote device through the socket. This call blocks
            // until it succeeds or throws an exception.
            mmSocket.connect();
        } catch (IOException connectException) {
            // Unable to connect; close the socket and return.
            try {
                mmSocket.close();
            } catch (IOException closeException) {
                Log.e(TAG, "Could not close the client socket", closeException);
            }
            return;
        }

        // The connection attempt succeeded. Perform work associated with
        // the connection in a separate thread.
        manageMyConnectedSocket(mmSocket);
    }
```

Bluetooth Implementation steps (4)

1. Setting up Bluetooth
2. Finding devices
3. Connecting devices
4. Managing a Connection
   - Simply reading/sending data using the socket
     - Get the InputStream and OutputStream and use getInputStream() and getOutputStream()
     - Read and write data using read(byte[]) and write(byte[]).

```
public ConnectedThread(BluetoothSocket socket) {
    mmSocket = socket;
    InputStream tmpIn = null;
    OutputStream tmpOut = null;

    // Get the input and output streams; using temp objects because
    // member streams are final.
    try {
        tmpIn = socket.getInputStream();
    } catch (IOException e) {
        Log.e(TAG, "Error occurred when creating input stream", e);
    }
    try {
        tmpOut = socket.getOutputStream();
    } catch (IOException e) {
        Log.e(TAG, "Error occurred when creating output stream", e);
    }

    mmInStream = tmpIn;
    mmOutStream = tmpOut;
}

public void run() {
    mmBuffer = new byte[1024];
    int numBytes; // bytes returned from read()

    // Keep listening to the InputStream until an exception occurs.
    while (true) {
        try {
            // Read from the InputStream.
            numBytes = mmInStream.read(mmBuffer);
            // Send the obtained bytes to the UI activity.
            Message readMsg = mHandler.obtainMessage(
                    MessageConstants.MESSAGE_READ, numBytes, -1,
                    mmBuffer);
            readMsg.sendToTarget();
        } catch (IOException e) {
            Log.d(TAG, "Input stream was disconnected", e);
            break;
        }
    }
}
```

Bluetooth: working with profiles

- Way simpler than before!

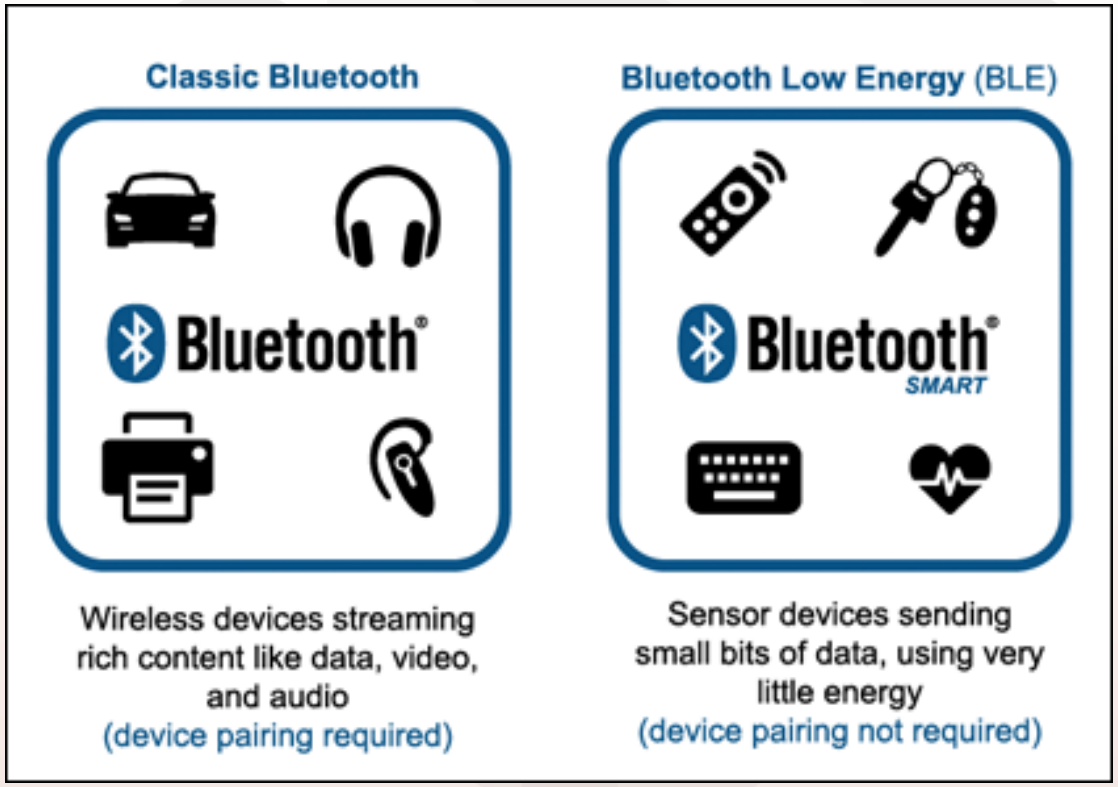
1. Get the default adapter
2. Set up a BluetoothProfile
   - notifies clients when they have been connected/disconnected
3. Use getProfileProxy() to establish a connection to the profile proxy object associated with the profile.
4. In onServiceConnected(), get a handle to the profile proxy object.
5. Monitor the state of the connection and perform other operations

Headset Profile Example:

```
BluetoothHeadset mBluetoothHeadset;

// Get the default adapter
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

private BluetoothProfile.ServiceListener mProfileListener = new BluetoothProfile.ServiceListener() {
    public void onServiceConnected(int profile, BluetoothProfile proxy) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = (BluetoothHeadset) proxy;
        }
    }
    public void onServiceDisconnected(int profile) {
        if (profile == BluetoothProfile.HEADSET) {
            mBluetoothHeadset = null;
        }
    }
};

// Establish connection to the proxy.
mBluetoothAdapter.getProfileProxy(context, mProfileListener, BluetoothProfile.HEADSET);

// ... call functions on mBluetoothHeadset

// Close proxy connection after use.
mBluetoothAdapter.closeProfileProxy(mBluetoothHeadset);
```

Bluetooth Low Energy (BLE)

- BLE is designed to provide significantly lower power consumption
- Apps can communicate with BLE devices that have low power requirements (proximity sensors, fitness/medical devices, etc.)
- Supported since Android 4.3 (API Level 18)
- The BLE API allows to:
  - Discover devices
  - Query for services
  - Send/receive data

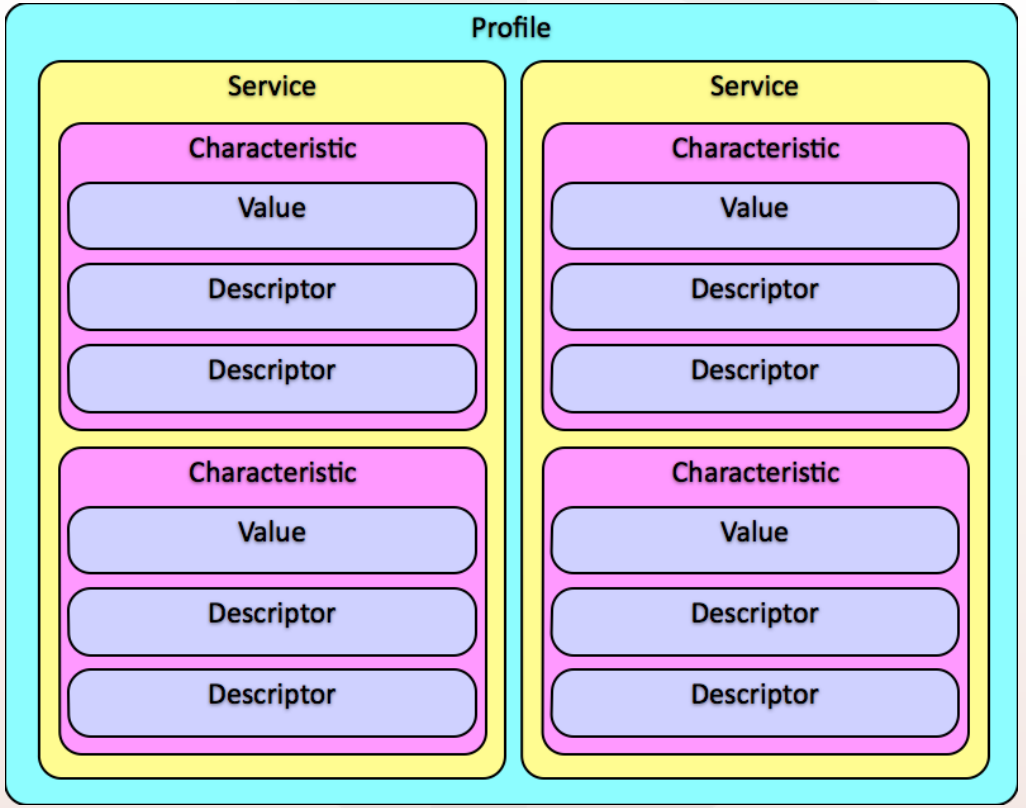


Generic Attribute Profile (GATT)

- General specification for sending and receiving short pieces of data known as "attributes" over a BLE link
  - All current Low Energy application profiles are based on GATT
  - Several profiles:
    - Battery level, Device information, Heart rate, Proximity
    - Complete list: https://developer.bluetooth.org/TechnologyOverview/Pages/Profiles.aspx
- GATT is built on top of the Attribute Protocol (ATT)
  - ATT is optimized to run on BLE devices (it uses as few bytes as possible)
  - Each attribute is identified by a Universally Unique Identifier (UUID)
- The attributes transported by ATT are formatted as characteristics and services

Characteristics and Services

- Characteristic
  - Contains a single value and several descriptors that describe the characteristic's value
  - A characteristic can be thought of as a type, analogous to a class.
- Descriptor
  - Describes a characteristic's value, for example:
    - A human-readable description
    - An acceptable range for a characteristic's value
    - The unit of measure of a characteristic's value
- Service
  - Collection of characteristics
  - For example, a service called "Heart Rate Monitor" that includes characteristics such as "heart rate measurement"
  - List of existing GATT-based profiles and services on bluetooth.org

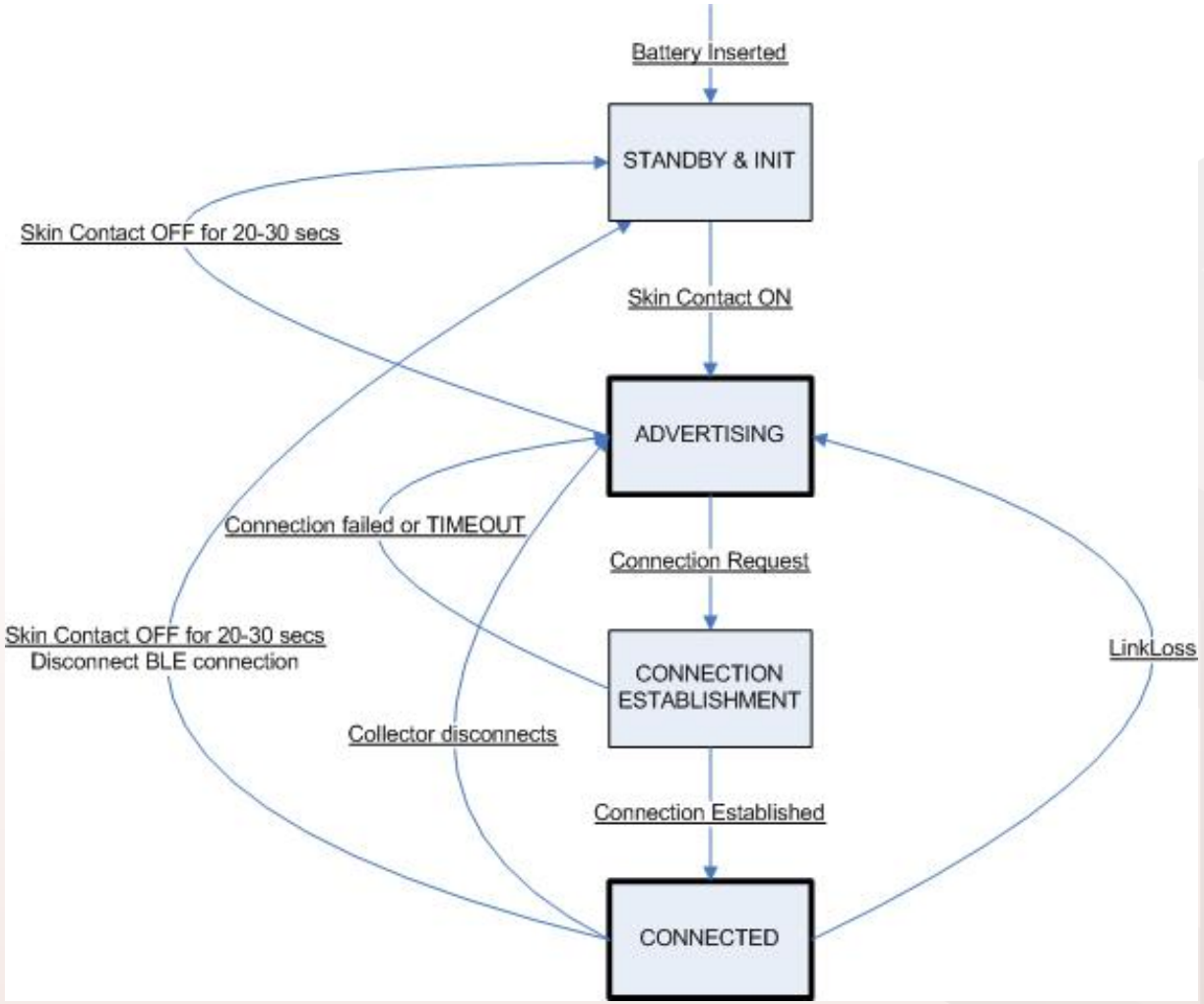


Polar H7 – Services

- Heart rate
  - Heart rate measurement
    - Value: “Heart rate measurement: 61bpm, Contact is Detected, RR Interval 983.04ms”
    - Descriptor: “Notifications enabled”
  - Body Sensor Location
    - Value: “Chest”
- Device information
  - Strings providing information about the device:
    - System ID, Model Number, Serial Number, Firmware Revision, Hardware Revision, Software Revision, Manufacturer Name
- Battery Service
  - Battery level
    - Value: “90%”

More info:
http://developer.polar.com/wiki/H6,_H7,_H10_and_OH1_Heart_rate_sensors

Polar H7 – BLE HR state machine



More info: http://developer.polar.com/wiki/H6,_H7,_H10_and_OH1_Heart_rate_sensors

Roles and Responsibilities

- Central vs. peripheral
  - Applies to the BLE connection itself
  - The device in the central role scans, looking for advertisement, and the device in the peripheral role makes the advertisement
- GATT server vs. GATT client
  - How two devices talk to each other once they've established the connection
- Example (Today's Lab 8):
  - Tablet → central device & GATT client
  - Polar H7 → peripheral device & GATT server

BLE permissions

- Declare the Bluetooth permissions in your application manifest file:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

- If you want to declare that your app is available to BLE-capable devices only, include the following in your app's manifest:

```
<uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
```

Setting Up BLE

- Get the **BluetoothAdapter**:

```
private BluetoothAdapter mBluetoothAdapter;
...
// Initializes Bluetooth adapter.
final BluetoothManager bluetoothManager =
        (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
mBluetoothAdapter = bluetoothManager.getAdapter();
```

- Enable Bluetooth:
  - Check whether Bluetooth is currently enabled
  - If it isn't, display an error prompting the user to go to Settings to enable it

```
// Ensures Bluetooth is available on the device and it is enabled. If not,
// displays a dialog requesting user permission to enable Bluetooth.
if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

Finding BLE Devices

- Scanning is battery-intensive, you should observe the following guidelines:
  - As soon as you find the desired device, stop scanning
  - Set a time limit on your scan. A device that was previously available may have moved out of range, and continuing to scan drains the battery
- Use **startLeScan**
- To scan for only specific types of peripherals, you can provide an array of UUID objects

```
public class DeviceScanActivity extends ListActivity {

    private BluetoothAdapter mBluetoothAdapter;
    private boolean mScanning;
    private Handler mHandler;

    // Stops scanning after 10 seconds.
    private static final long SCAN_PERIOD = 10000;
    ...
    private void scanLeDevice(final boolean enable) {
        if (enable) {
            // Stops scanning after a pre-defined scan period.
            mHandler.postDelayed(new Runnable() {
                @Override
                public void run() {
                    mScanning = false;
                    mBluetoothAdapter.stopLeScan(mLeScanCallback);
                }
            }, SCAN_PERIOD);

            mScanning = true;
            mBluetoothAdapter.startLeScan(mLeScanCallback);
        } else {
            mScanning = false;
            mBluetoothAdapter.stopLeScan(mLeScanCallback);
        }
        ...
    }
...
}
```

Connecting to a GATT Server

- Connect:
  - BluetoothGatt connectGatt (Context context, boolean autoConnect, BluetoothGattCallback callback)
- Discover services:
  - mBluetoothGatt.discoverServices()
- Read BLE attributes
  - Read and write attributes. For example, display services and characteristics and display them in the UI

Receiving GATT Notifications

- The app can be notified when a particular characteristic changes on the device

- Enable notification for a characteristic:
  - setCharacteristicNotification()

- An onCharacteristicChanged() callback is triggered if the characteristic changes on the remote device

```
private BluetoothGatt mBluetoothGatt;
BluetoothGattCharacteristic characteristic;
boolean enabled;
...
mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
...
BluetoothGattDescriptor descriptor = characteristic.getDescriptor(
        UUID.fromString(SampleGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));
descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
mBluetoothGatt.writeDescriptor(descriptor);
```

```
@Override
// Characteristic notification
public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic) {
    broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
}
```

Closing the Client App

- After the app has finished using the device, we close it:

```
public void close() {
    if (mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.close();
    mBluetoothGatt = null;
}
```

- More information about Android BLE API
  - http://developer.android.com/guide/topics/connectivity/bluetooth-le.html

Outline of the class

- Connecting to the internet: WiFi and 3G/4G
  - Transferring data without draining your battery

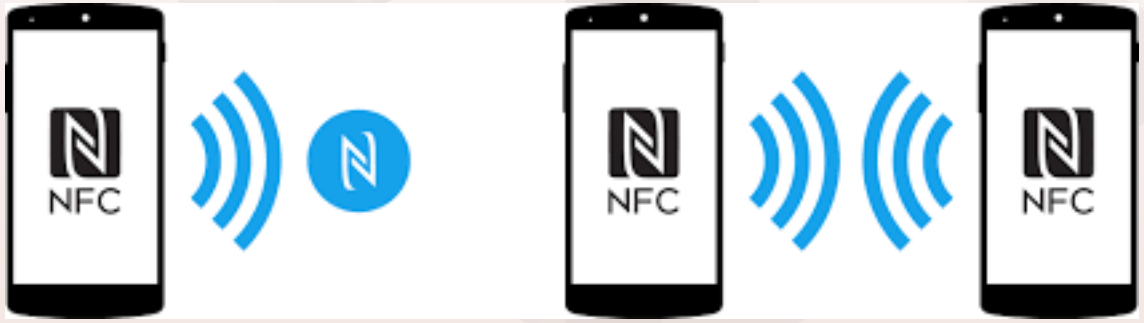
- Bluetooth
  - Bluetooth Low Energy
  - Polar H7

- **NFC**

Near Field Communication (NFC)

- NFC is a set of short-range wireless technologies, usually below 4cm
  - Share small set of data between an NFC tag and an Android device
  - … or between two Android-powered devices.

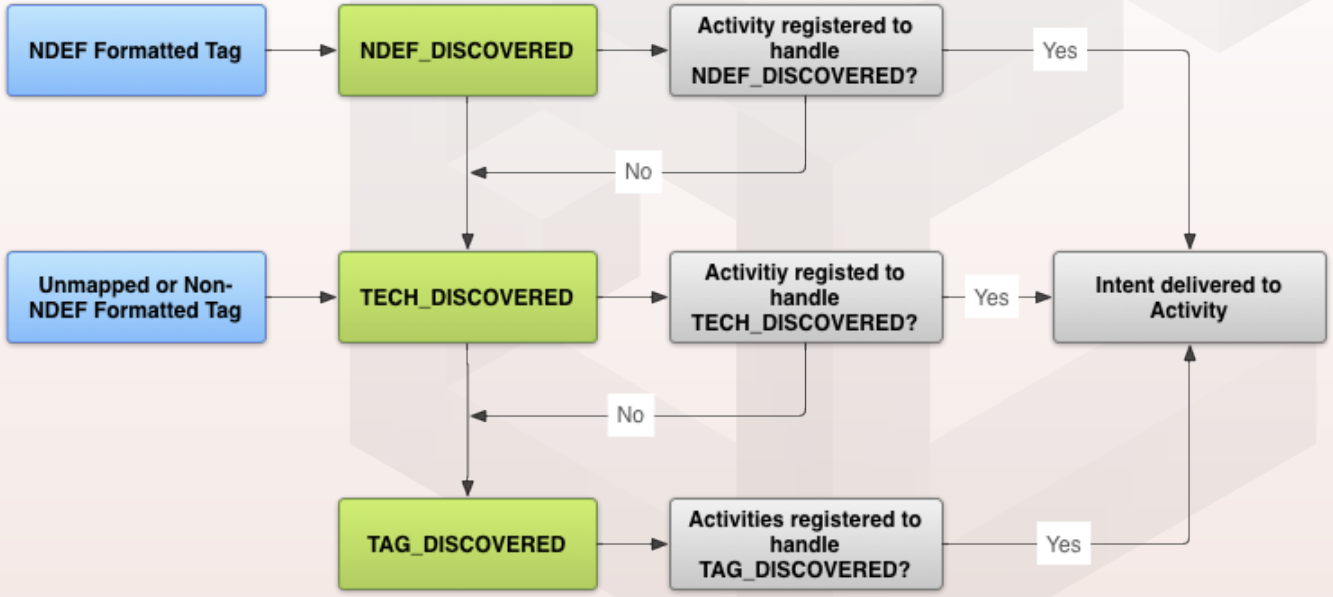
- Three main modes of operation:
  - **Reader/writer mode**: the NFC device will read/write passive NFC tags and stickers.
  - **P2P mode**: the NFC device will exchange data with other NFC peers
  - **Card emulation mode**, allowing the NFC device itself to act as an NFC card.
    - The emulated NFC card can then be accessed by an external NFC reader, such as an NFC point-of-sale terminal.



The tag dispatch system

- Reading an NFC tag from Android:
  - Android provides an API to read/write data stored in a tag, using the NDEF (NFC Data Exchange Format) standard
  - Unless NFC is disabled, Android is always looking for NFC tags when the screen is unlocked

- The Tag dispatch system takes care of:
  - Analyzes scanned NFC tags and parses them
  - Sends an intent to an interested application that filters the intent
    - Prioritized list of Intents!



Steps to read from NFC tag

1. Requesting NFC permissions
2. Filtering for NFC intents in an Activity
3. Obtaining information about the tag from the Intent

1. Manifest.xml

```
<uses-permission android:name="android.permission.NFC" />
```

2.

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain" />
</intent-filter>
```

3.

```
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    ...
    if (intent != null && NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction())) {
        Parcelable[] rawMessages =
            intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
        if (rawMessages != null) {
            NdefMessage[] messages = new NdefMessage[rawMessages.length];
            for (int i = 0; i < rawMessages.length; i++) {
                messages[i] = (NdefMessage) rawMessages[i];
            }
            // Process the messages array.
            ...
        }
    }
}
```

Beaming NDEF messages to other devices

- Android allows you simple peer-to-peer data exchange between two devices.
  - Application sending must be in the foreground
  - Device receiving must not be locked

- You can enable **AndroidBeam** from your app calling:
  - setNdefPushMessage(): Accepts an NdefMessage to set as the message to beam. Automatically beams the message when two devices are in close enough proximity.
  - Or setNdefPushMessageCallback(): Accepts a callback that contains a createNdefMessage() which is called when a device is in range to beam data to.
    The callback lets you create the NDEF message only when necessary.

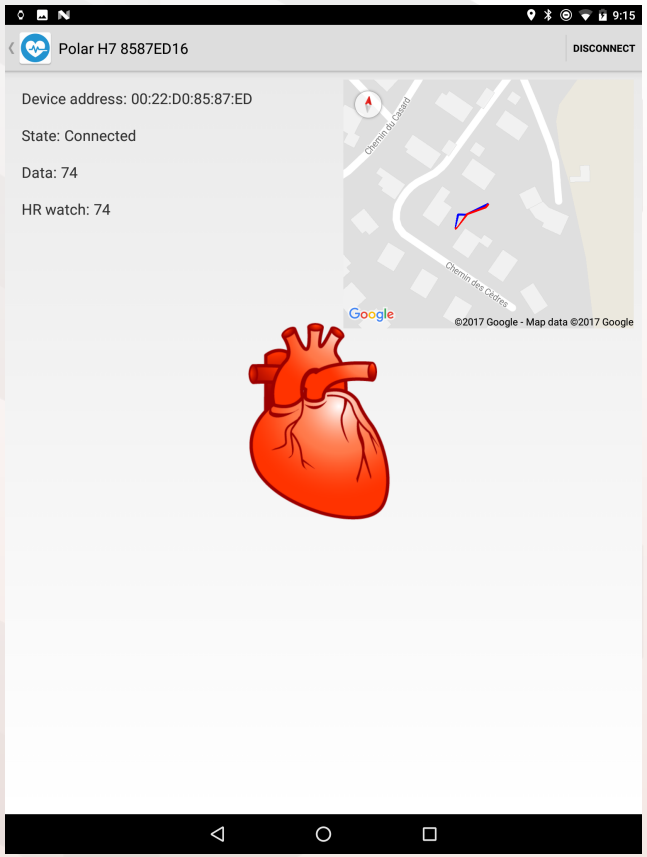
AndroidBeam example

```
public class Beam extends Activity implements CreateNdefMessageCallback {
    NfcAdapter mNfcAdapter;
    TextView textView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView textView = (TextView) findViewById(R.id.textView);
        // Check for available NFC Adapter
        mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
        if (mNfcAdapter == null) {
            Toast.makeText(this, "NFC is not available", Toast.LENGTH_LONG).show();
            finish();
            return;
        }
        // Register callback
        mNfcAdapter.setNdefPushMessageCallback(this, this);
    }
```

```
    @Override
    public NdefMessage createNdefMessage(NfcEvent event) {
        String text = ("Beam me up, Android!\n\n" +
                "Beam Time: " + System.currentTimeMillis());
        NdefMessage msg = new NdefMessage(
                new NdefRecord[] { createMime(
                        "application/vnd.com.example.android.beam", text.getBytes())
         /**
          * The Android Application Record (AAR) is commented out. When a device
          * receives a push with an AAR in it, the application specified in the AAR
          * is guaranteed to run. The AAR overrides the tag dispatch system.
          * You can add it back in to guarantee that this
          * activity starts when receiving a beamed message. For now, this code
          * uses the tag dispatch system.
          */
          //,NdefRecord.createApplicationRecord("com.example.android.beam")
        });
        return msg;
    }

    @Override
    public void onResume() {
        super.onResume();
        // Check to see that the Activity started due to an Android Beam
        if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
            processIntent(getIntent());
        }
    }
```

Today's lab

- Connecting to the Polar H7 via Bluetooth Low Energy
- Using the GoogleMaps API
- Understanding the AndroidWear communication
  - Which is using Services



What next?

- Next sessions:
  - No lectures or labs → Time for you to work on your projects
- Projects:
  - Next week: Tue. Nov. 20th → final project assignment
    - Check who your TA is!
    - The TA will answer all questions/doubts related to your project
  - Initial exam timeslot assignment by Nov. 27th
- Mid-term exam → **December 11th**
  - Questions session: December 4th

Questions?