App Development for Smart Devices

CS 495/595 – Fall 2012

Lec #8: Networking & SMS

Tamer Nadeem
Dept. of Computer Science
Objective

• SMS/MMS
  • Using Intents to send
  • Using SMS Manager
  • Handling incoming SMS

• Bluetooth
  • Managing Bluetooth Properties
  • Device Discovery
  • Bluetooth Communication

• Network Connectivity & WiFi

• Student Presentations
  – Using Idle Moments to Record Your Health via Mobile Applications
    • Presenter: Lulwah Alkwai
  – Detecting driver phone use leveraging car speakers
    • Presenter: Sarwar Sha
  – How Long to Wait?: Predicting Bus Arrival Time with Mobile Phone based Participatory Sensing
    • Presenter: Thomas Carson
SMS and MMS
Overview

• SMS sends short text messages between mobile phones.
  ➤ Supports sending both text messages and data messages

• MMS (multimedia messaging service) messages have allowed users to send and receive messages that include multimedia attachments such as photos, videos, and audio.

• Using the SMSManager, you can replace the native SMS application to send text messages, react to incoming texts, or use SMS as a data transport layer.

• Use the SEND and SEND_TO actions in Intents to send both SMS and MMS messages using a messaging application installed on the device.
Sending SMS/MMS thru Native App

- Use Intent with `Intent.ACTION_SENDTO` action:
  - Specify a target number using `sms: schema` notation as the Intent data.
  - Include the message you want to send within the Intent payload using an `sms_body` extra.

```java
Intent smsIntent = new Intent(Intent.ACTION_SENDTO, Uri.parse("sms:55512345"));
smsIntent.putExtra("sms_body", "Press send to send me");
startActivity(smsIntent);
```
Sending SMS/MMS thru Native App

• You can also attach files (effectively creating an MMS message) to your messages
  ➤ Add an `Intent.EXTRA_STREAM` with the URI of the resource to attach.
  ➤ Set the Intent `type` to the `mime-type` of the attached resource.
  ➤ Use `ACTION_SEND` and include the target phone number as an address extra

```java
// Get the URI of a piece of media to attach.
Uri attached_Uri = Uri.parse("content://media/external/images/media/1");

// Create a new MMS intent
Intent mmsIntent = new Intent(Intent.ACTION_SEND, attached_Uri);
mmsIntent.putExtra("sms_body", "Please see the attached image");
mmsIntent.putExtra("address", "07912355432");
mmsIntent.putExtra(Intent.EXTRA_STREAM, attached_Uri);
mmsIntent.setType("image/png");
startActivity(mmsIntent);
```
Sending SMS Manually

- SMS messaging in Android is handled by the `SmsManager`.
  ```java
  SmsManager smsManager = SmsManager.getDefault();
  ```

- Specify the `SEND_SMS` uses-permission.
  ```xml
  <uses-permission android:name="android.permission.SEND_SMS"/>
  ```

- Use `sendTextMessage` from the SMS Manager, passing in the address (phone number) of your recipient and the text message you want to send,
  ```java
  String sendTo = "5551234";
  String myMessage = "Android supports programmatic SMS messaging!";
  smsManager.sendTextMessage(sendTo, null, myMessage, null, null);
  ```

**SMS Manager Reference:**
Tracking and Confirming SMS Delivery

• The final two parameters in `sendTextMessage` let you specify Intents to track the transmission and delivery.

• Implement and register corresponding Broadcast Receivers that listen for the actions you specify when creating the Pending Intents you pass in `sendTextMessage`.

• Intent parameter, `sentIntent`, is fired when the message either is successfully sent or fails to send.
  ➤ Activity.RESULT_OK
  ➤ SmsManager.RESULT_ERROR_GENERIC_FAILURE
  ➤ SmsManager.RESULT_ERROR_RADIO_OFF
  ➤ SmsManager.RESULT_ERROR_NULL_PDU

• The second Intent parameter, `deliveryIntent`, is fired only after the destination recipient receives your SMS message.
**SMS delivery monitoring pattern**

```java
String SENT_SMS_ACTION = "SENT_SMS_ACTION";
String DELIVERED_SMS_ACTION = "DELIVERED_SMS_ACTION";

// Create the sentIntent parameter
Intent sentIntent = new Intent(SENT_SMS_ACTION);
PendingIntent sentPI = PendingIntent.getBroadcast(getApplicationContext(),
                                            0, sentIntent, 0);

// Create the deliveryIntent parameter
Intent deliveryIntent = new Intent(DELIVERED_SMS_ACTION);
PendingIntent deliverPI = PendingIntent.getBroadcast(getApplicationContext(),
                                           0, deliveryIntent, 0);

// Register the Broadcast Receivers
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context _context, Intent _intent) {
        switch (getResultCode()) {
            case Activity.RESULT_OK:
                [ . . . send success actions . . . ]; break;
            case SmsManager.RESULT_ERROR_GENERIC_FAILURE:
                [ . . . generic failure actions . . . ]; break;
        }
    }
});
```
case SmsManager.RESULT_ERROR_RADIO_OFF:
    [ . . . radio off failure actions . . . ]; break;
    case SmsManager.RESULT_ERROR_NULL_PDU:
    [ . . . null PDU failure actions . . . ]; break;
}
}
}
,
new IntentFilter(SENT_SMS_ACTION));

registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context _context, Intent _intent) {
        [ . . . SMS delivered actions . . . ]
    }
},
new IntentFilter(DELIVERED_SMS_ACTION));

// Send the message
smsManager.sendTextMessage(sendTo, null, myMessage, sentPI, deliverPI);
Large SMS Messages

- SMS text messages are normally limited to 160 characters.
- Longer messages need to be broken into a series of smaller parts.
  - `divideMessage` method accepts a string as an input and breaks it into an Array List of messages
  - Use the `sendMultipartTextMessage` method on the SMS Manager to transmit the array of messages
  - The `sentIntent` and `deliveryIntent` parameters in the `sendMultipartTextMessage` method are Array Lists that is used to specify different Pending Intents to fire for each message part.

```java
ArrayList<String> messageArray = smsManager.divideMessage(myMessage);
ArrayList<PendingIntent> sentIntents = new ArrayList<PendingIntent>();
for (int i = 0; i < messageArray.size(); i++)
    sentIntents.add(sentPI);

smsManager.sendMultipartTextMessage(sendTo, null,
    messageArray, sentIntents, null);
```
Handling Incoming SMS Messages

• With received SMS, new broadcast Intent is fired with the “android.provider.Telephony.SMS_RECEIVED“ action.

• Specify the RECEIVE_SMS manifest permission.

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

• Use the pdu extras key to extract an array of SMS PDUs each of which represents an SMS message

• Call SmsMessage.createFromPdu to convert each PDU byte array into an SMS Message object

```
Bundle bundle = intent.getExtras();
if (bundle != null) {
    Object[] pdus = (Object[]) bundle.get("pdus");
    SmsMessage[] messages = new SmsMessage[pdus.length];
    for (int i = 0; i < pdus.length; i++)
        messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
}
```
Example of Incoming SMS Messages

- Register the Broadcast Receiver using an Intent Filter that listens for the android.provider.Telephony.SMS_RECEIVED action String

```java
final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
IntentFilter filter = new IntentFilter(SMS_RECEIVED);
BroadcastReceiver receiver = new IncomingSMSReceiver(); //defined below
registerReceiver(receiver, filter);
```

- Broadcast Receiver implementation whose onReceive handler checks incoming SMS texts that start with the string @echo, and then sends the same text back to the number that sent it.

```java
public class IncomingSMSReceiver extends BroadcastReceiver {
    private static final String queryString = "@echo";
    private static final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
```
Example of Incoming SMS Messages

```java
public void onReceive(Context _context, Intent _intent) {
    if (_intent.getAction().equals(SMS_RECEIVED)) {
        SmsManager sms = SmsManager.getDefault();
        Bundle bundle = _intent.getExtras();
        if (bundle != null) {
            Object[] pdus = (Object[]) bundle.get("pdus");
            SmsMessage[] messages = new SmsMessage[pdus.length];
            for (int i = 0; i < pdus.length; i++)
                messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

            for (SmsMessage message : messages) {
                String msg = message.getMessageBody();
                String to = message.getOriginatingAddress();

                if (msg.toLowerCase().startsWith(queryString)) {
                    String out = msg.substring(queryString.length());
                    sms.sendTextMessage(to, null, out, null, null);
                }
            }
        }
    }
}
```
Simulating Incoming SMS Messages/Calls

• Use the Android debug tools to simulate incoming SMS messages or calls from arbitrary numbers.

![Emulator Control Interface for Simulating SMS Messages/Calls]
Bluetooth
Overview

• Bluetooth is a specification for the use of low power wireless communications over short distance.

• Although Bluetooth standard utilizes the same 2.4 GHz range of Wi-Fi

• Compared to Wi-Fi, Bluetooth networking is slower, a bit more limited in range, and supports many fewer devices
Bluetooth States
Android Bluetooth

- Classes support Bluetooth devices and connections:
  - **BluetoothAdapter** represents the local Bluetooth device on which your application is running.
  - **BluetoothDevice** Each remote device with which you wish to communicate.
  - **BluetoothSocket** let you make a connection request to the remote device, and then initiate communications.
  - **BluetoothServerSocket** use it on your local Bluetooth Adapter to listen for incoming connection requests from Bluetooth Sockets on remote devices.
Bluetooth Device Adapter

- To access the default Bluetooth adapter on the host device call **getDefaulAdapter**.
  
  ```java
  BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();
  ```

- To read any of the local Bluetooth Adapter properties, initiate discovery, or find bonded devices: include the **BLUETOOTH** manifest permission.

- To modify any of the local device properties: include the **BLUETOOTH_ADMIN** manifest permission.

  ```xml
  <uses-permission android:name="android.permission.BLUETOOTH"/>
  <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
  ```
Managing Bluetooth Properties

- Reading/Changing properties needs Bluetooth adapter to be on
- Access the Bluetooth Adapter’s *friendly name* (an arbitrary string that users can set) and hardware address

```java
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();

String toastText;
if (bluetooth.isEnabled()) {
    String address = bluetooth.getAddress();
    String name = bluetooth.getName();
    toastText = name + " : " + address;
} else
    toastText = "Bluetooth is not enabled";

Toast.makeText(this, toastText, Toast.LENGTH_LONG).show();
```

- Change the friendly name of the Bluetooth Adapter using:

```java
bluetooth.setName("Blackfang");
```
Enabling Bluetooth Adaptor

- By default the Bluetooth adapter will be turned off
- Enable the Bluetooth Adapter thru system sub-Activity using the ACTION_REQUEST_ENABLE
  - Use the result code parameter returned in the onActivityResult handler to determine the success of this operation

```java
String enableBT = BluetoothAdapter.ACTION_REQUEST_ENABLE;
startActivityForResult(new Intent(enableBT), 0);
```

- You can turn the Bluetooth Adapter on and off directly, using the enable and disable methods.
Enabling Bluetooth Adaptor

- Enabling/disabling are time-consuming, asynchronous operations.
  - Register a Broadcast Receiver that listens for `ACTION_STATE_CHANGED`.
  - The broadcast Intent will include two extras, `EXTRA_STATE` and `EXTRA_PREVIOUS_STATE`, the current and previous states.

```java
BluetoothAdapter bluetooth = BluetoothAdapter.getDefaultAdapter();

BroadcastReceiver bluetoothState = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String prevStateExtra = BluetoothAdapter.EXTRA_PREVIOUS_STATE;
        String stateExtra = BluetoothAdapter.EXTRA_STATE;
        int state = intent.getIntExtra(stateExtra, -1);
        int previousState = intent.getIntExtra(prevStateExtra, -1);

        String tt = "";
        switch (state) {
            case (BluetoothAdapter.STATE_TURNING_ON) : {
                tt = "Bluetooth turning on"; break;
            }
        }
    }
};
```
Enabling Bluetooth Adaptor

case (BluetoothAdapter.STATE_ON) : {
    tt = "Bluetooth on";
    unregisterReceiver(this);
    break;
}
case (BluetoothAdapter.STATE_TURNING_OFF) : {
    tt = "Bluetooth turning off"; break;
}
case (BluetoothAdapter.STATE_OFF) : {
    tt = "Bluetooth off"; break;
    default: break;
}

Toast.makeText(this, tt, Toast.LENGTH_LONG).show();

};

if (!bluetooth.isEnabled()) {
    String actionStateChanged = BluetoothAdapter.ACTION_STATE_CHANGED;
    String actionRequestEnable = BluetoothAdapter.ACTION_REQUEST_ENABLE;
    registerReceiver(bluetoothState,
        new IntentFilter(actionStateChanged));
    startActivityForResult(new Intent(actionRequestEnable), 0);
}
Device Discovery

• The process of two devices finding each other in order to connect is called *discovery*.
  ➢ Before you can establish a Bluetooth Socket for communications, the local Bluetooth Adapter must bond with the remote device.
  ➢ Before two devices can bond and connect, they first need to discover each other.

• In order for remote Android Devices to find your local Bluetooth Adapter during a discovery scan, you need to ensure that it is discoverable.
Managing Device Discoverability

- The Adapter’s discoverability is indicated by its scan mode.
- Call `getScanMode` on the `BluetoothAdapter` object. It returns:
  - `SCAN_MODE_CONNECTABLE_DISCOVERABLE` Inquiry Scan and Page Scan are both enabled → device is discoverable.
  - `SCAN_MODE_CONNECTABLE` Page Scan is enabled but Inquiry Scan is not → devices that have previously connected and bonded to the local device can find it during discovery, but new devices can’t.
  - `SCAN_MODE_NONE` Discoverability is turned off → No remote devices can find the local adapter during discovery.

- By default, discoverability is disabled. To turn on discovery use `start activity` with `ACTION_REQUEST_DISCOVERABLE`:

```java
String aDiscoverable = BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE;
startActivityForResult(new Intent(aDiscoverable), DISCOVERY_REQUEST);
```
Managing Device Discoverability

- To handle user response, override `onActivityResult` handler.
  ➤ The returned `resultCode` indicates the duration of discoverability.

```java
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == DISCOVERY_REQUEST) {
        boolean isDiscoverable = resultCode > 0;
        int discoverableDuration = resultCode;
    }
}
```

- Monitor `ACTION_SCAN_MODE_CHANGED` broadcast action.
  ➤ Broadcast Intent has current and previous scan modes as extras.

```java
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String prevScanMode = BluetoothAdapter.EXTRA_PREVIOUS_SCAN_MODE;
        String scanMode = BluetoothAdapter.EXTRA_SCAN_MODE;
        int scanMode = intent.getIntExtra(scanMode, -1);
        int prevMode = intent.getIntExtra(prevScanMode, -1);
    }
}, new IntentFilter(BluetoothAdapter.ACTION_SCAN_MODE_CHANGED));
```
Discovering Remote Devices

• The discovery process can take some time to complete (up to 12 seconds).

• To check if the Adapter is already performing a discovery scan, use the `isDiscovering` method.

• To initiate the discovery process call `startDiscovery` on the Bluetooth Adapter. To cancel a discovery in progress call `cancelDiscovery`.

• The discovery process is asynchronous. Android uses broadcast Intents to notify you of the start (`ACTION_DISCOVERY_STARTED`) and end (`ACTION_DISCOVERY_FINISHED`) of discovery as well as remote devices discovered (`ACTION_FOUND`) during the scan.
BroadcastReceiver discoveryMonitor = new BroadcastReceiver() {
    String dStarted = BluetoothAdapter.ACTION_DISCOVERY_STARTED;
    String dFinished = BluetoothAdapter.ACTION_DISCOVERY_FINISHED;

    @Override
    public void onReceive(Context context, Intent intent) {
        if (dStarted.equals(intent.getAction())) {
            // Discovery has started.
            Toast.makeText(getApplicationContext(),
            "Discovery Started . . . ", Toast.LENGTH_SHORT).show();
        }
        else if (dFinished.equals(intent.getAction())) {
            // Discovery has completed.
            Toast.makeText(getApplicationContext(),
            "Discovery Completed . . . ", Toast.LENGTH_SHORT).show();
        }
    }
};

registerReceiver(discoveryMonitor, new IntentFilter(dStarted));
registerReceiver(discoveryMonitor, new IntentFilter(dFinished));
Discovering remote Bluetooth Devices

• Each broadcast Intent includes the name of the remote device in an extra `BluetoothDevice.EXTRA_NAME`, and representation of the remote device under `BluetoothDevice.EXTRA_DEVICE` extra.

```java
BroadcastReceiver discoveryResult = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String remoteDeviceName = intent.getStringExtra(BluetoothDevice.EXTRA_NAME);
        BluetoothDevice remoteDevice;

        remoteDevice = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        Toast.makeText(getApplicationContext(), "Discovered: " + remoteDeviceName, Toast.LENGTH_SHORT).show();
        // TODO Do something with the remote Bluetooth Device.
    }
};
registerReceiver(discoveryResult, new IntentFilter(BluetoothDevice.ACTION_FOUND));
if (!bluetooth.isDiscovering())
    bluetooth.startDiscovery();
```
Bluetooth Communications

- The Bluetooth communications APIs are wrappers around RFCOMM, the Bluetooth radio frequency communications protocol.
  - RFCOMM supports RS232 serial communication over the Logical Link Control and Adaptation Protocol (L2CAP) layer.

- You must have a **client** and a **server**. Used classes are:
  - **BluetoothServerSocket**: Used to establish a listening socket at the **server** for initiating a link between devices.
  - **BluetoothSocket**: Used in creating a new **client** socket to connect to a listening Bluetooth Server Socket, and returned by the Server Socket once a connection is established.

- Once the connection is made, Bluetooth Sockets are used on both the **server** and **client** sides to transfer data streams.

- Connection is done in **separate thread**
RFCOMM (Radio Frequency Communication)

- The Bluetooth protocol RFCOMM is a simple set of transport protocols.
- RFCOMM is sometimes called Serial Port Emulation.
- The Bluetooth Serial Port Profile is based on this protocol.

- In the protocol stack, RFCOMM is bound to L2CAP
- RFCOMM provides a simple reliable data stream to the user, similar to TCP. It is used directly by many telephony related profiles as a carrier for AT commands
Bluetooth Server

• Use `BluetoothServerSocket` to `listen` for incoming connection requests get by calling `listenUsingRfcommWithServiceRecord`
  ➤ Passing: String “`name`” to identify your server, and
  ➤ UUID (universally unique identifier) to be used by clients to connect

• To start listening, call `accept` on this Server Socket
  ➤ Optionally passing in a timeout duration.

• The Server Socket will `block` until a remote Bluetooth Socket client with a matching UUID attempts to connect.

• If a remote device is not yet paired, the user will be prompted to accept a pairing request before the accept call returns.

• If an incoming connection request is successful, `accept` will return a `Bluetooth Socket` connected to the client device.
public class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {
        BluetoothServerSocket tmp = null;
        try {
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
        } catch (IOException e) { ... }
        mmServerSocket = tmp;
    }

    public void run() {
        BluetoothSocket socket = null;
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                break;
            }
            if (socket != null) {
                // You must implement this method
doSomethingWithTheConnection(socket);
                mmServerSocket.close();
                break;
            }
        }
    }

    public void cancel() {
        try {
            mmServerSocket.close();
        } catch (IOException e) { }
    }
}
Bluetooth Client

- Use **BluetoothSocket** to create a new **connection** to the server by calling **createRfcommSocketToServiceRecord**
  - Passing: UUID of the Bluetooth Server Socket accepting requests

- If you attempt to connect to a Bluetooth Device that has not yet been paired (bonded) with current host, you will be prompted to accept the pairing before the connect call completes.

- The user must accept the pairing request on both the host and remote devices for the connection to be established.

- The returned Bluetooth Socket can then be used to initiate the connection with a call to connect.
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        BluetoothSocket tmp = null;
        mmDevice = device;

        try {
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) { ... }
        mmSocket = tmp;
    }

    public void run() {
        mBluetoothAdapter.cancelDiscovery();

        try {
            mmSocket.connect();
        } catch (IOException connectException) {
            try {
                mmSocket.close();
            } catch (IOException closeException) { }
            return;
        }

        manageConnectedSocket(mmSocket);
    }

    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}
Network Connectivity
Network Connectivity

• Different network options (Wi-Fi, GPRS, 3G) have different characteristics (speed, reliability, cost) in accessing Internet.

• Your applications should be able to know and manage these connections to ensure they run efficiently and responsively.

• Android networking is principally handled via the `ConnectivityManager`.

• `WifiManager` lets you monitor and control the Wi-Fi connectivity.

• Android broadcasts Intents that describe changes in network connectivity and offers APIs to control network settings.
Connectivity Manager

- Use `ConnectivityManager` service to
  - Monitor the connectivity state
  - Set your preferred network connection
  - Manage connectivity failover.

```java
String service = Context.CONNECTIVITY_SERVICE;
ConnectivityManager connectivity = (ConnectivityManager) getSystemService(service);
```

- Need to enable read and write network state access permissions.

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

More Details on `ConnectivityManager`:
Background Data Transfer

• User sets preference for background data transfers.
  ➤ Enable/Disable background data transfers: Settings ➡ Accounts & sync settings ➡ Background data setting.
  ➤ This value is enforced at the application level

• Obtain the background data setting through calling `getBackgroundDataSetting` on the `ConnectivityManager`.

```java
boolean backgroundEnabled = connectivity.getBackgroundDataSetting();
```

• If the background data setting is disabled
  ➤ Your application should transfer data only when it is active and in the foreground.
  ➤ User explicitly requests that your application not transfer data when it is not visible and in the foreground.
Background Data Transfer

- If your application requires background data transfer, notify users and offer to go to the settings page to enable it.

- When user changes the background data preference, the system sends a broadcast Intent with the `ConnectivityManager.ACTION_BACKGROUND_DATA_SETTING_CHANGED` action.

- Register a new Broadcast Receiver that listens for this Intent

```java
registerReceiver(
    new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // Do something when the background data setting changes.
        },
    new IntentFilter(ConnectivityManager.ACTION_BACKGROUND_DATA_SETTING_CHANGED)));
```
Monitoring Network Details

• ConnectivityManager provides info on the available connections

• Use `getActiveNetworkInfo` or `getNetworkInfo` methods to return `NetworkInfo` object

➤ To find the connection status, network type, and detailed state information of the returned network

```java
// Get the active network information.
NetworkInfo activeNetwork = connectivity.getActiveNetworkInfo();
int networkType = networkInfo.getType();
switch (networkType) {
    case (ConnectivityManager.TYPE_MOBILE) : break;
    case (ConnectivityManager.TYPE_WIFI) : break;
    default: break;
}

// Get the mobile network information.
int network = ConnectivityManager.TYPE_MOBILE;
NetworkInfo mobileNetwork = connectivity.getNetworkInfo(network);
NetworkInfo.State state = mobileNetwork.getState();
NetworkInfo.DetailedState detailedState = mobileNetwork.getDetailedState();
```

More Details on `NetworkInfo`:
Controlling Hardware Radios

• When application requests an Internet connection, Android attempts to connect using the preferred network.

• Use `getNetworkPreference` and `setNetworkPreference` to find the current, and set the preferred, network.

```java
int networkPreference = connectivity.getNetworkPreference();
connectivity.setNetworkPreference(NetworkPreference.PREFER_WIFI);
```

• Use `setRadio` to control the availability of the network types.

```java
connectivity.setRadio(NetworkType.WIFI, false);
connectivity.setRadio(NetworkType.MOBILE, true);
```
Monitoring Network Connectivity

- To monitor network connectivity, create, listen for broadcasted `ConnectivityManager.CONNECTIVITY_ACTION` Intents. Include several extras to additional details on the change.
  - `EXTRA_IS_FAILOVER` - true if the current connection is the result of a failover from a preferred network.
  - `EXTRA_NO_CONNECTIVITY` - device is not connected
  - `EXTRA_REASON` - string describing why the connection failed.
  - `EXTRA_NETWORK_INFO` - returns `NetworkInfo` object with details.
  - `EXTRA_OTHER_NETWORK_INFO` - returns `NetworkInfo` object populated with details for the possible failover network connection.
  - `EXTRA_EXTRA_INFO` - contains additional network-specific extra connection details.
WiFi
WiFi Manager

• The **WifiManager** represents the Android Wi-Fi Connectivity Service. Used to:
  ➤ Configure Wi-Fi network connections
  ➤ Manage the current Wi-Fi connection
  ➤ Scan for access points
  ➤ Monitor changes in Wi-Fi connectivity.

• Use **Context.WIFI_SERVICE** constant to access WifiManager

```java
String service = Context.WIFI_SERVICE;
WifiManager wifi = (WifiManager) getSystemService(service);
```

• Permission to access/change the Wi-Fi state

```xml
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
```

More Details on WifiManager:
MANAGING YOUR WI-FI

• Enable/Disable Wi-Fi hardware using the `setWifiEnabled` method

• Request current Wi-Fi state using the `getWifiState` or `isWifiEnabled` methods

```java
if (!wifi.isWifiEnabled())
    if (wifi.getWifiState() != WifiManager.WIFI_STATE_ENABLING)
        wifi.setWifiEnabled(true);
```

• WifiManager provides low-level access to the Wi-Fi network configurations.
  ➤ Full control over each Wi-Fi configuration setting to completely replace the native Wi-Fi management application if required.
Monitoring Wi-Fi Connectivity

- The Wi-Fi Manager broadcasts one of the following Intents with the change in connectivity status:
  
  ➤ **WIFI_STATE_CHANGED_ACTION** - moves between enabling, enabled, disabling, disabled, and unknown. Includes two extras **EXTRA_WIFI_STATE** and **EXTRA_PREVIOUS_STATE** for the new and previous states.
  
  ➤ **SUPPLICANT_CONNECTION_CHANGE_ACTION** - the connection state with the active supplicant (access point) changes. It is fired when a new connection is established or an existing connection is lost, using the **EXTRA_NEW_STATE** Boolean extra, which returns true in the former case.
Monitoring Wi-Fi Connectivity

- Continue the broadcasts Intents:
  - ➤ **NETWORK_STATE_CHANGED_ACTION** - the Wi-Fi connectivity state changes. Two extras: `EXTRA_NETWORK_INFO` includes a `NetworkInfo` object that details the current network state, and `EXTRA_BSSID` includes the BSSID of the access point you’re connected to.
  - ➤ **RSSI_CHANGED_ACTION** - monitor the current signal strength of the connected Wi-Fi network. Includes extra `EXTRA_NEW_RSSI` that holds the current signal strength.
    - To use this signal strength you should use the `calculateSignalLevel` static method on the Wi-Fi Manager to convert it to an integer value on a scale you specify.
Monitor Active Connection

- Use the `getConnectionInfo` method on the `WifiManager` to find info on the active connection’s status.

  ➤ The returned `WifiInfo` object includes SSID, BSSID, MAC address, and IP address of the current access point, as well as the current link speed and signal strength.

```java
WifiInfo info = wifi.getConnectionInfo();
if (info.getBSSID() != null) {
    int strength = WifiManager.calculateSignalLevel(info.getRssi(), 5);
    int speed = info.getLinkSpeed();
    String units = WifiInfo.LINK_SPEED_UNITS;
    String ssid = info.getSSID();
    String cSummary = String.format("Connected to %s at %s. Strength %s/5", ssid, speed, units, strength);
}
```

More Details on `WifiInfo`:
Scanning for Hotspots

• Wi-Fi Manager can conduct access point scans using the **startScan** method.
  
  ➤ An Intent with the **SCAN_RESULTS_AVAILABLE_ACTION** action will be broadcast to asynchronously announce that the scan is complete and results are available.
  
  ➤ Call **getScanResults** to get those results as a list of **ScanResult** objects.
  
  ➤ Each Scan Result includes the details retrieved for each access point detected, including link speed, signal strength, SSID, and the authentication techniques supported.

• Following example shows how to initiate a scan for access points that displays a Toast indicating the total number of access points found and the name of the access point with the strongest signal.
// Register a broadcast receiver that listens for scan results.
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        List<ScanResult> results = wifi.getScanResults();
        ScanResult bestSignal = null;
        for (ScanResult result : results) {
            if (bestSignal == null ||
                WifiManager.compareSignalLevel(bestSignal.level, result.level) < 0)
                bestSignal = result;
        }

        String toastText = String.format("%s networks found. %s is the strongest.",
            results.size(), bestSignal.SSID);

        Toast.makeText(getApplicationContext(), toastText, Toast.LENGTH_LONG);
    }
}, new IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));

// Initiate a scan.
wifi.startScan();
Creating Wi-Fi Network Configurations

• Create and register a configuration to connect to a Wi-Fi network.

• Network configurations are stored as WifiConfiguration objects.

• The following is a non-exhaustive list of some of the public fields available for each Wi-Fi configuration:
  • ➤ BSSID - The BSSID for an access point
  • ➤ SSID - The SSID for a particular network
  • ➤ networkId - unique identifier to identify this network configuration
  • ➤ priority - priority when ordering list of access points to connect to
  • ➤ status - The current status of this network connection, which will be one of the following: WifiConfiguration.Status.ENABLED, WifiConfiguration.Status.DISABLED, or WifiConfiguration.Status.CURRENT
Creating Wi-Fi Network Configurations

• The configuration object also contains the supported authentication techniques, as well as the keys used previously to authenticate with this access point.

• The `addNetwork` method lets you specify a new configuration to add to the current list.

• The `updateNetwork` lets you update a network configuration by passing in a WifiConfiguration with a network ID and the values you want to change.

• You can also use `removeNetwork`, passing in a network ID, to remove a configuration.

• To persist any changes made to the network configurations, you must call `saveConfiguration`. 
Managing Wi-Fi Configurations

• Once connected, use WiFi Manager to interrogate the active network connection to get additional details of its configuration and settings.

• Use `getConfiguredNetworks` for current network configurations list
  ➤ The list of `WifiConfiguration` objects returned includes the network ID, SSID, and other details for each configuration.

• To use particular network configuration, use the `enableNetwork` method, passing in the network ID to use and specifying true for the `disableAllOthers` parameter

```java
// Get a list of available configurations
List&lt;WifiConfiguration&gt; configurations = wifi.getConfiguredNetworks();
// Get the network ID for the first one.
if (configurations.size() &gt; 0) {
    int netID = configurations.get(0).networkId;
    // Enable that network.
    boolean disableAllOthers = true;
    wifi.enableNetwork(netID, disableAllOthers);
}
```
Questions?